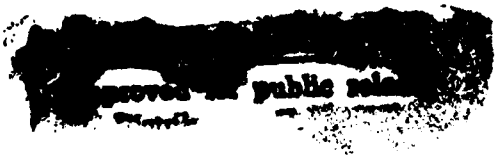
USING THE CoRE REQUIREMENTS METHOD WITH ADARTS℠

SPC-93091-CMC

VERSION 01.00.05

MARCH 1994



REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1994	Technical Report

4. TITLE AND SUBTITLE

Using CoRE Requirements Method with ADARTS

5. FUNDING NUMBERS

G MDA972-92-J-1018

6. AUTHOR(S) H. Lykins, R. Kirk, D. Smith
Produced by Software Productivity Consortium under contract to Virginia Center of Excellence

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES)
Virginia Center of Excellence
SPC Building
2214 Rock Hill Road
Herndon, VA 22070

8. PERFORMING ORGANIZATION REPORT NUMBER

SPC-93091-CMC,
Version 01.00.03

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
ARPA/SISTO
Suite 400
801 N. Randolph Street
Arlington, VA 22203

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The CoRE Guidebook is available from DTIC (ADA # 274691). The ADARTS Guidebook (SPC-94040-CMC) may be obtained from Software Productivity Consortium by calling (703) 742-7211.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

No Restrictions

12b. DISTRIBUTION CODE

1

13. ABSTRACT *(Maximum 200 words)*

This technical report explains how you can use the Ada-Based Design Approach for Real-Time Systems (ADARTS^{SM}) to build a software design that satisfies software requirements specified using the Consortium Requirements Engineering Method (CoRE). ADARTS is a widely accepted object-oriented method for system and software development that results in a robust design that is well documented, meets timing requirements, can withstand change, and contains many reusable components. CoRE is a new object-oriented approach to software requirements engineering that results in requirements that are precise, testable, complete, consistent, and resilient in the face of change. This report is a supplement to the ADARTS Guidebook (Version 2) and the CoRE Guidebook (Version 1) and discusses: developing a CoRE requirements specification for use with ADARTS; deriving an ADARTS process structure from CoRE requirements; combining ADARTS processes and objects derived from CoRE requirements into an ADARTS software architecture design; and taking advantage of CoRE's precision in the ADARTS process structuring, class structuring, and software architecture design activities.

14. SUBJECT TERMS
Object-oriented requirements and design, concurrency, real-time software, ADARTS, CoRE, formal specification, evaluation criteria

15. NUMBER OF PAGES
182

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

USING THE CoRE REQUIREMENTS METHOD WITH ADARTS®

SPC-93091-CMC

VERSION 01.00.05

MARCH 1994

Howard Lykins
Richard A. Kirk
Doug Smith

Produced by the
SOFTWARE PRODUCTIVITY CONSORTIUM SERVICES CORPORATION
under contract to the
VIRGINIA CENTER OF EXCELLENCE
FOR SOFTWARE REUSE AND TECHNOLOGY TRANSFER

SPC Building
2214 Rock Hill Road
Herndon, Virginia 22070
(703) 742–7211

CONTENTS

FIGURES

TABLES

ACKNOWLEDGMENTS

The authors and the Consortium wish to thank those who contributed to the development of this technical report:

- James Kirby, Jr. who helped develop an early version of the ideas in this report and who reviewed the draft of the final version.
- Other Consortium staff who reviewed the draft of this report: Lisa Finneran, Kent Johnson, and Steve Wartik. Their comments have helped to make this report what it is.
- Other members of the ADARTS team (Christine Ausnit and Mike Cochran), the CoRE team (Stuart Faulk, Assad Moini, and Skip Osborne), plus Ron Damer and Neil Burkhard, all of whom have reviewed and commented on the design approach in this report.
- Jim Sutton, Mike Sullivan, and other members of the Lockheed C-130J software team, the first industrial project to use CoRE and ADARTS for software development.
- Mary Mallonee for technical editing; Debbie Morgan for entering the markups to the draft document; and Betty Leach and Tina Medina for clean proofing the final document.

This page intentionally left blank.

EXECUTIVE SUMMARY

The purpose of this technical report is to explain how to take requirements specified using the Consortium Requirements Engineering (CoRE) method and develop a software design using the heuristics and guidelines from the Ada-based Design Approach for Real-Time Systems (ADARTS℠). This technical report is the result of more than a year of research, development, and pilot project activity directed toward integrating CoRE and ADARTS. It is part of an ongoing effort to integrate and improve the products of the Software Productivity Consortium.

Combined Benefits of Both Methods

CoRE is a new approach to software requirements engineering that results in requirements that are precise, testable, complete, consistent, and resilient in the face of change. ADARTS is a widely accepted object-oriented method for system and software development that results in a robust design that is well documented, meets timing requirements, can withstand change, and contains many reusable components. By using ADARTS and CoRE together, you obtain the benefits of both methods.

Increased Precision of Software Design

A major benefit of using ADARTS with CoRE is the increased precision of ADARTS work products. The precision of CoRE's behavioral model will enable you to precisely specify the behavior of design components, facilitating verification and minimizing the risk of misunderstanding by implementors and customers. This technical report contains optional enhanced verification guidelines for two ADARTS software design activities, based on the increased precision.

Similarity of Concepts

ADARTS and CoRE have many concepts in common, eliminating the need for a "paradigm shift" when moving from requirements specification to design. Software engineers have an easier time transitioning from requirements analysis to design if the two activities are based on similar concepts. Because both ADARTS and CoRE use object-oriented concepts, the transition from one activity to another is smoother than it often has been in the past.

Pilot Project Validation

The first pilot project to use ADARTS with CoRE, Lockheed's avionics redesign for the C-130J, was conducted in parallel with the development of this report. This pilot provided useful feedback to the Consortium, resulting in improved guidelines for CoRE and for the use of ADARTS with CoRE.

What Is in This Technical Report

This technical report explains how to develop an ADARTS software design that satisfies a CoRE requirements specification. It is intended to be used as a supplement to the *Consortium Requirements*

Engineering Guidebook (version 01.00.09) and *ADARTS Guidebook* (version 02.00.13) and provides guidance in two areas:

1. ADARTS software design guidelines that must change to be used with CoRE requirements
2. ADARTS software design guidelines that should change to benefit from the increased precision provided by CoRE requirements

Guidelines in the second category are optional; engineers do not have to follow them to use ADARTS with CoRE.

Where CoRE has no impact on ADARTS design activities, engineers will use the heuristics in the *ADARTS Guidebook*.

1. INTRODUCTION

1.1 PURPOSE OF THIS TECHNICAL REPORT

This technical report explains how you can use the Ada-Based Design Approach for Real-Time Systems (ADARTS℠) to build a software design to satisfy software requirements specified using the Consortium Requirements Engineering Method (CoRE). This report is intended to be used as a supplement to the *ADARTS Guidebook*, version 02.00.13 (Software Productivity Consortium 1991), herein called the ADARTS Guidebook, and *Consortium Requirements Engineering Guidebook*, version 01.00.09 (Software Productivity Consortium 1993), herein called the CoRE Guidebook, and discusses the following:

- Developing a CoRE requirements specification for use with ADARTS
- Deriving an ADARTS process structure from CoRE requirements
- Deriving an ADARTS class structure from CoRE requirements
- Combining ADARTS processes and objects derived from CoRE requirements into an ADARTS software architecture design
- Taking advantage of CoRE's precision in the ADARTS process structuring, class structuring, and software architecture design activities

1.2 INTENDED AUDIENCE

This technical report is directed at technologists and engineers who are very familiar with the ADARTS and CoRE methods. This technical report does not attempt to explain either ADARTS or CoRE; it assumes that you are comfortable with each.

1.3 HOW TO USE THIS TECHNICAL REPORT

Section 3 provides a brief supplement to the CoRE Guidebook and discusses how you apply CoRE to build a software requirements specification for ADARTS. Subsequent sections supplement chapters in Volume 1 of the ADARTS Guidebook. Each major subsection in this technical report identifies the section or subsection of the ADARTS Guidebook that it supplements. This technical report provides guidance in two areas:

1. Areas in which design activities must change to be used with CoRE requirements
2. Areas in which design activities should change to benefit from the increased precision provided by CoRE requirements

Where CoRE has no impact on ADARTS design activities, you should follow what is stated in the ADARTS Guidebook. This technical report addresses ADARTS software design activities. It does not discuss system-level design.

1.4 ORGANIZATION OF THIS REPORT

This technical report is organized as follows:

- ***Introduction.*** Sections 1 and 2 introduce the report and provide an overview of how you use ADARTS with CoRE. Section 2 contains important information about the assumptions and basic approach to design used in this report.
- ***Requirements Specification.*** Section 3 describes how you use CoRE to build a software requirements specification for use with ADARTS.
- ***Process Structuring.*** Section 4 describes how you derive ADARTS processes from CoRE requirements and how you take advantage of CoRE's precision to make clustering decisions and to specify and evaluate the process architecture.
- ***Class Structuring.*** Section 5 describes how you derive ADARTS classes and objects from CoRE requirements and how you take advantage of CoRE's precision to specify and evaluate the classes.
- ***Software Architecture Design.*** Section 6 describes how you combine a process architecture and class structure derived from a CoRE requirements specification into an ADARTS software architecture design.
- ***Case Study.*** The Appendix provides a case study that illustrates the guidelines in Sections 3, 4, 5, and 6.

1.5 TYPOGRAPHIC CONVENTIONS

This report uses the following typographic conventions:

Serif font	General presentation of information.
Italicized serif font	Mathematical expressions and publication titles.
Boldfaced serif font	Section headings and emphasis.
Boldfaced italicized serif font	Run-in headings in bulleted lists and, in the Appendix, minor subsections.
`Typewriter` font	ADARTS class specifications.
{ }	Definition of a set or bag.
[ ]	Optional items (zero or one).
|	Separator for a list of alternatives.

2. OVERVIEW OF DESIGN APPROACH

This section provides an overview of the approach described in this report to building an ADARTS software design from CoRE requirements and documents the fundamental assumptions underlying the approach. You should read this section before reading subsequent sections.

2.1 THE DESIGN APPROACH

Figure 1 illustrates the ADARTS approach to designing software from requirements expressed in CoRE. The activities and dependencies between activities are the same as in ADARTS. After completing your requirements specification, you perform the ADARTS process structuring and class structuring activities. These two activities can be performed concurrently or sequentially in arbitrary order.

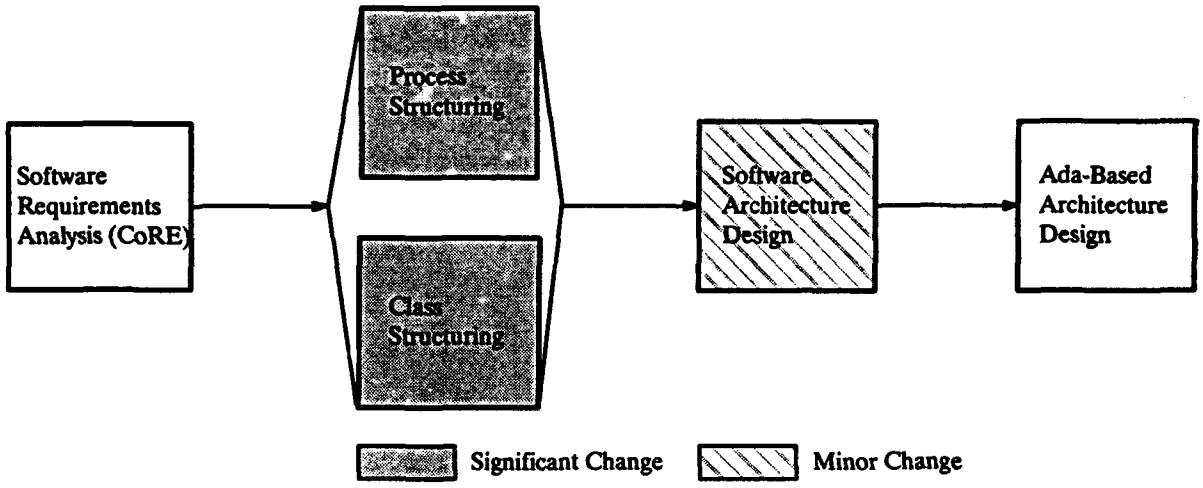


Figure 1. ADARTS Software Development Activities

In process structuring, you develop the dynamic view of the software architecture, concentrating on concurrency, sequencing, and timing. First, you follow the guidelines in this report to create the initial process architecture. You then follow the ADARTS Guidebook to cluster processes. In class structuring, you develop the static view of the software, concentrating on encapsulation, information hiding, and planning for change. This report tells you how to map CoRE requirements to ADARTS classes and objects. In software architecture design, you merge the results of process and class structuring into a unified software design. In Ada-based architecture design, you choose constructs in the Ada programming language for each element of the software architecture design.

This report provides guidance in the following areas: aspects of ADARTS that must change to create a design to satisfy CoRE requirements and aspects that should change to benefit from the precision

of CoRE requirements. Guidelines in the second category are optional; you do not have to follow them, but you will have a more precise design if you do. The optional guidelines are enhancements to ADARTS facilitated by the precision of CoRE. The only identified changes to software architecture design are enhancements. There are no changes to Ada-based architecture design discussed in this report.

2.2 TERMINOLOGY

The CoRE behavioral model is in terms of four types of variables (monitored, controlled, input, and output) and four relations between them (required [REQ], nature [NAT], input [IN], and output [OUT]). The CoRE relations other than NAT contain value functions that usually appear as tables in a CoRE specification. The CoRE value functions specify one of the following mappings:

- Monitored variables to controlled variables (REQ relation)
- Monitored variables to input variables (IN relation)
- Output variables to controlled variables (OUT relation)

CoRE augments value functions with nonzero bounds on error and delay, which makes REQ, IN, and OUT relations rather than functions. Although the CoRE Guidebook uses the term "value function" only in reference to the REQ relation, this report uses the term "CoRE value function" to refer to tables defining any of the three mappings described above.

2.3 NECESSARY CHANGES TO PROCESS STRUCTURING

The necessary changes to ADARTS process structuring are:

1. The requirements artifacts you use to develop the initial process architecture
2. The heuristics you use to determine the need for data storage

The guidelines for data storage are necessary because a CoRE requirements specification contains references to past values of variables instead of defining data stores. Guidelines for process clustering and evaluation of the process architecture do not have to change from ADARTS, although this report describes optional enhancements to both steps.

Figure 2 shows the structure of the initial process architecture. Processes in the initial process architecture are motivated by events in the CoRE specification. The mapping of requirements to the initial process architecture described in this report serves two purposes: it is straightforward and is intended to allow you to take full advantage of potential concurrency. The final process architecture, which results from your application of the clustering criteria, will almost certainly have fewer processes. The following discussion explains the significance of each kind of process:

- An input stimulus (IN_s) process responds to an input stimulus and retrieves the value of an input variable from a device.
- An input translation (IN_t) uses the input variable retrieved by the IN_s process to approximate the value of the corresponding monitored variable. The tilde ("~") signifies that the value

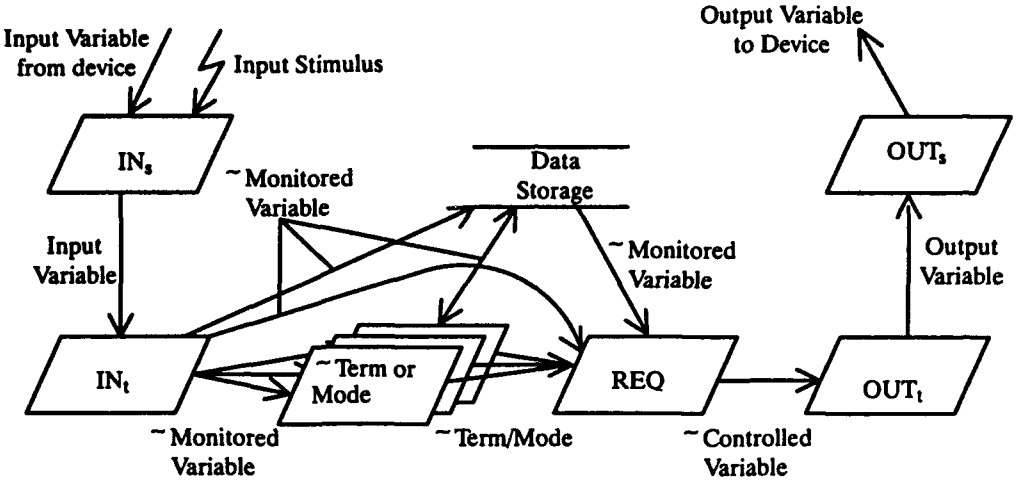


Figure 2. Schematic for Initial Process Architecture

computed by the process is an approximation. The approximation of the monitored variable is used by Term, Mode, or REQ processes that need it.

- There is one Term process for each term defined using a CoRE event. A Term process receives an input, which is an approximation of a monitored variable or term, or a mode and determines if the event defining the term has occurred.

- There is one Mode process for each mode machine in the CoRE specification. A Mode process uses an input, which is an approximation of a monitored variable or term, or a mode of another mode machine and determines if a mode transition has occurred.

- A REQ process uses an input, which is an approximation of a monitored variable or term, or a mode of a mode machine and updates its approximation of the controlled variable. A stimulus/response thread that causes a visible change will always go through a REQ process.

- An OUT_t process uses an approximation of a controlled variable and generates the appropriate value of an output variable. Again, stimulus/response threads that cause visible change will always go through an OUT_t process.

- At the required times, an OUT_s process sends output variable value(s) to the associated device. Stimulus/response threads that cause visible change will always go through an OUT_t process.

The purpose of IN_s and IN_t processes is to respond to changes in input variables. Term and mode processes are motivated by events referenced by mode machine and term definitions. REQ processes are motivated by changes in monitored variables and the need to change the corresponding controlled variable. OUT_s and OUT_t to set output variables so that the corresponding controlled variables are set properly.

A CoRE specification does not include requirements for internal data storage. Instead, it references past values of variables that the designer translates into a need for data storage. Figure 2 illustrates that IN_t and Term processes may save approximations of monitored variables and terms for future use.

2.4 NECESSARY CHANGES TO CLASS STRUCTURING

As with process structuring, the necessary changes to class structuring are limited to the requirements artifacts that you map to classes and objects. Guidelines for abstract interfaces, the generalization/specialization structure, the dependency graph, the information hiding structure, and evaluation criteria do not have to change, although this report describes optional enhancements for the abstract interface and evaluation criteria. Table 1 summarizes how you map CoRE requirements to ADARTS classes and objects.

2.5 OPTIONAL ENHANCEMENTS

This report discusses a number of optional enhancements to the process structuring, class structuring, and software architecture design activities. All of these enhancements are facilitated by the precision of CoRE requirements and are motivated by the desire to maintain CoRE's level of precision during design. You do not have to take advantage of the enhancements discussed in this report. However, your design will benefit from precise specification if you do so. The benefits of precision include:

- Lack of ambiguity, which decreases the probability of misunderstanding by implementors and reviewers
- Improved guidelines for work product evaluation, leading to greater confidence in the design and reducing the probability of errors.

Enhancements to process structuring include improved guidelines for periodic temporal clustering and evaluation criteria. Enhancements to class structuring include improved guidelines for work product evaluation. Precise notations for specifying process and class behavior are introduced in the appropriate sections. These notations permit enhanced evaluation criteria for the software architecture design activity as well. In addition, software architecture design is enhanced with guidelines for relating delay and error.

2.6 CONCERNS TO REMEMBER

This section discusses a number of concerns you should keep in mind while applying the guidelines in this technical report. These concerns, along with the overviews of the process and class structuring activities, are the motivation for the design approach described in this technical report.

2.6.1 CoRE Variables and ADARTS Approximations

The design approach in this report is based on software approximations of environmental quantities. The monitored and controlled variables in a CoRE specification, as well as terms and modes, represent quantities in the environment. The only variables that the software can observe and set directly are input and output variables. For example, software cannot directly observe a monitored variable, such as the level of fuel in a tank. However, it is possible for software to approximate the value of a monitored variable, deriving the approximation from input variable(s) retrieved from the hardware. For example, the software can approximate the level of fuel in a tank by reading from an input device the amount of pressure exerted by the fuel or the position of a float.

Table 1. Derivation of Classes and Objects

Kind of Class	CoRE Element	Basis of Operation	Objects Created
Device Interface	Hardware devices (or groups of similar devices) described by input and output relations IN and OUT Relations CoRE boundary classes	Reading input variables Writing output variables Possibly approximating monitored variables and deriving output variables	One for each device
External System	Requirements for which you will use external systems IN and OUT Relations CoRE Boundary Classes	Controlling and communicating with an external system	One for each external system
Data Abstraction	Monitored and controlled variables Input and output variables Terms Expressions in REQ, IN, and OUT tables CoRE boundary, term, and mode classes	Reading, comparing, and setting internal approximations of values and performing mathematical operations on the approximations	One for each variable or term
Collection	Monitored and controlled variables Input and output variables Terms Expressions in REQ, IN, and OUT tables CoRE boundary and term classes	Operations on a set of values (e.g., Create, Destroy, Add, Delete, Iterate, Search, Compare, Retrieve, Copy, etc.)	One for each variable or term defined as a collection of values
State Transition	Each unique mode machine CoRE mode classes	An operation for each CoRE event that causes a mode change, or operations for groups of events	One for each state transition class One for each mode machine if identical mode machines mapped to the same class
User Interface	Look and feel requirements IN and OUT relations CoRE boundary classes	Operations for acquiring information from and providing information to human users	One or more for each user interface class
Computation	Tables: REQ, IN, OUT relations, term definitions Complex expressions within tables CoRE boundary, term, and mode classes	One for each way in which the computation can be invoked	One for each computation class if there is no internal state, possibly multiple objects if there is an internal state

It is essential that you remember the difference between a monitored variable and the software's approximation of it. In almost all cases, the software's approximation will differ from the monitored variable because of the inaccuracy inherent in computer arithmetic and because the monitored variable can be changing while the software is approximating its value. The same observation applies to terms, modes, and controlled variables. Because they are ultimately defined from monitored variables, the software can only approximate their values. This technical report denotes an approximation with the tilde ("~"). For example, ~mon_Fuel_Level would be the software's approximation of the monitored variable mon_Fuel_Level. It is strongly recommended that you use this notation or a similar notation to distinguish approximations from the real variables.

In CoRE, delay and error values are used with the ideal functions to describe behavior. In the case of REQ, they describe the tolerable behavior and, in that context, can be called tolerances. In the case of IN and OUT, they describe the worst case delay and error that the software must assume during design and, in that context, describe the precision of the devices. In developing an ADARTS design, you should convince yourself that your software sets the values of controlled variables within the tolerance and delay specified in REQ. To do this, you will have to consider the delay and imprecision of input and output devices and delay and error introduced by software.

2.6.2 CoRE Events and ADARTS Stimuli

All CoRE events signify changes in environmental variables. Environmental variables include time, a monitored variable. You use CoRE events to determine the need for ADARTS processes and the stimuli that cause them to respond. This is one of the more difficult aspects of design and one place for you to apply engineering judgment. The frequency of the event, behavior of the device, and (sometimes) the maximum rate of change for the monitored variables can all influence the designer's choice of ADARTS stimuli, message communication, and process logic.

In contract to CoRE events, ADARTS events may be external events or timer events. External events signify hardware interrupts, and timer events signify the passage of time. The difference between CoRE events and ADARTS events is that CoRE events refer to changes in environmental quantities and ADARTS events refer to changes that the software can observe directly.

Section 4 uses the term "unique event." When a CoRE event table is used to define a value function, there are usually several event expressions. Often, these event expressions have annotations, indicating that the behavior associated with the event applies only in a specific mode or when a specific condition holds. Regardless of the annotations, you should treat multiple event expressions in an event table that describe the same event as a single, unique event. For example, "@T(mon_Temperature < 0)" describes the same event as "@F(mon_Temperature $\geq$ 0)." For purposes of ADARTS process structuring, the expressions "@T(mon_Temperature<0)" and "@T(mon_Temperature<0) when inmode(mode_Normal)" initiate the same stimulus-response thread, although the latter expression identifies a qualifying condition for responding to the event. The process logic of the process responding to an event with a qualifying condition must specify how behavior differs under each different condition.

2.6.3 Use of the CoRE Value Functions

The value function of the CoRE IN relation maps monitored variables to input variables, and the value function of the OUT relation maps output variables to controlled variables. In the design approach

described in this report, the inverse of these functions is necessary. Given an input variable, your design will approximate the corresponding monitored variable, and given an approximation of a controlled variable, your design will calculate the appropriate value of an output variable. The notation IN' is used in this report to refer to the inversion of the IN value function with monitored variables replaced by their approximations. The notation OUT' is used in this report to refer to the inversion of the OUT value function, with controlled variables replaced by their approximations.

The value function of the CoRE REQ relation maps monitored variables and/or terms to a controlled variable. This value function need not be inverted for design, but the value function expressed in terms of ADARTS objects (i.e., software variables) is useful for ADARTS process and class structuring. The notation REQ' is used in this report to refer to the REQ value function, with monitored and controlled variables and terms replaced by their approximations.

REQ', IN', and OUT' are functions derived from REQ, IN, and OUT, respectively, and are used in this report to describe guidelines in process structuring, class structuring, and software architecture design. Table 2 summarizes the purposes of these derived functions.

Table 2. Derived Functions

Relation	Purpose	Derived Function	Purpose
REQ	Relates monitored variables to controlled variables	REQ'	Function that returns an approximation of a controlled variable given monitored variable and/or term approximation(s).
IN	Relates monitored variables to input variables	IN'	Function that returns an approximation of a monitored variable given an input variable approximation.
OUT	Relates output variables to controlled variables	OUT'	Function that returns an approximation of an output variable given controlled variable approximation(s).

The derived functions are useful during ADARTS process structuring when identifying stimuli that cause processes to respond and when identifying process logic. For example, REQ_for_con_Report describes which report is generated when the event_Periodic_60_Second occurs, depending upon the current mode. A process is added to the ADARTS initial process architecture diagram named Generate_Periodic_Reports that responds to event_Periodic_60_Second by generating the appropriate report. Derived functions are also used during class structuring to describe the abstract interface of classes.

Inverting value functions is not always trivial, so you may already have documented the inversion of IN and OUT value functions in the CoRE specification. It is not necessary to document REQ', IN', and OUT' as work products. However, both process structuring and class structuring use the inversion of IN and OUT value functions. Therefore, in the nontrivial cases, make sure that you invert the functions once to save time and avoid confusion before starting process and class structuring.

2.6.4 Use of the CoRE NAT Relation

The NAT relation documents constraints placed on the software system by the external environment and constraints on monitored and controlled variables. You should consider NAT when you are

defining the behavior of processes and classes. For example, in process structuring, you can use NAT to determine the frequency of a timer event based on knowledge about the maximum rate of change of a monitored variable and the required tolerance of a controlled variable. In class structuring, you can use NAT to determine parts of the abstract interface, such as assumptions, usage constraints, and undesired events.

2.6.5 Dealing With Delay and Error

The IN and OUT relations capture the worst case delay and error associated with input/output hardware. The REQ relation captures the maximum tolerance for error in a controlled variable and the initiation delay and completion deadline for setting a controlled variable's value. To meet requirements, the software must set the value of a controlled variable within the time interval defined by the initiation delay and completion deadline, and the value must be within the specified tolerance.

The easiest approach to dealing with delay and error is to consider them separately, dealing with delay during process structuring and error during class structuring. In process structuring, you should estimate the execution of each process and ensure that every controlled variable is set between the initiation delay and completion deadline, taking into consideration the delay imposed by hardware devices.

In class structuring, you should record the maximum error associated with operations and ensure that total error imposed by software will not cause the value of a controlled variable to exceed the tolerance in REQ. To do this, you should consider each class and operation needed to set a controlled variable from the monitored variable. This includes retrieving an input variable, using it to approximate a monitored variable, using that approximation to approximate the controlled variable, determining the appropriate value of the output variable, and sending the output variable value to the device. The total of the maximum error associated with each operation, combined with the device errors, must not exceed the error bound specified in the REQ relation.

If this is not feasible, you must consider the relationship between delay and error and take into account the rate of change of the monitored variable. This is discussed in more detail in Sections 3.3 and 6.4.

2.7 NOTATION

This section describes the notation used in many of the examples. The notation used in this report has the advantage of precision, which benefits you as described in Section 2.5 **However, a specific notation is not critical to the design approach described in this document.** Wherever possible when discussing requirements, this report follows the notation of the CoRE Guidebook. When dealing with design, this report follows the notation of the ADARTS Guidebook.

For elements of the design, the tilde denotes approximations of the environmental variables. For example, "~mon_Buoy_Location" represents an ADARTS design element that approximates the monitored variable mon_Buoy_Location.

REQ', IN', and OUT' are notations representing modified CoRE value functions and are defined in Section 2.6.3.

The subscripts $_s$ and $_t$ identify specific types of processes in the initial process architecture and are discussed in Section 2.3.

This report uses concepts and notation from set theory to introduce more precision in the descriptions of ADARTS work products. Braces are used to itemize the elements of a set. The empty set is denoted by "{}." Where S, S_1, and S_2 are sets or bags[1], this report uses the following standard operations from set theory:

- S_1 UNION S_2 denotes the set composed of all elements in either S_1 or S_2 or both.
- S_1 INTERSECT S_2 denotes the set composed of all elements in both S_1 and S_2.
- $S_1 - S_2$ is the set formed by removing from S_1 all elements in S_1 INTERSECT S_2 .
- SIZE(S) denotes the number of elements in S.

In addition, this report introduces the following nonstandard operations:

- OLDEST(S) denotes the oldest (i.e., least recently added) element of S.
- SUM(S) is the arithmetic sum of the elements in S. SUM(S) is undefined if the elements of S are not numeric.

The following notation is used to define the content of a set, where it is not feasible to itemize the elements. Where i is a placeholder, D(i) is an expression involving i, and F(i) is some mathematical function of i, the set {i: D(i): F(i)} is the set of all values F(i) such that D(i) holds. The placeholder i has no meaning outside the braces; it is similar in that respect to a local variable declared in a subroutine. D(i) defines the set from which i is taken; applying F to each value in this set produces the set defined by the expression. Unless otherwise noted, placeholders i, j, k, l, m, and n denote integers; other letters denote real numbers. For example:

- $\{i: i>0 : i^2\} = \{1, 4, 9, ...\}$ is the set of squares of the positive integers.
- $\{i: i>0: i\} = \{1, 2, 3, ...\}$ is the set of positive integers.
- $\{i: 0<i\leq 5: i\} = \{1,2,3,4,5\}$ is a finite set with five elements.

A similar notation is used to define operations on a set without defining the set separately. For example:

- $(SUM\ i: 0<i\leq 4: i^2) = 1+4+9+16=30$ is the sum of the squares of the first four integers.
- ROUND[(SUM (i:0≤i≤5:mon_Air_Temperature(t−10i)) / 6] is the average of the past six readings of air temperature, where the readings are taken at 10-second intervals and the most recent reading just occurred.

Logical conjunction, disjunction, and negation are denoted by AND, OR, and NOT, respectively.

The following notation is used to specify the abstract interfaces of classes. Where P is an expression, ERROR(P) in a postcondition means that an operation on a class returns an error flag or raises an exception signifying P. Where X is a name referenced in a precondition, Updated_X is used in a postcondition to denote the value of X between completion of the operation and the next change to X. X and Updated_X will usually refer to the abstract state.

1. A bag is a set in which elements can be repeated. Many collections of data stored by software are bags rather than sets.

This page intentionally left blank.

3. BUILDING A CoRE SPECIFICATION FOR USE WITH ADARTS

This section highlights the features of CoRE that deserve special attention when ADARTS is the companion design method. These suggestions should make the design activity simpler and reduce the need to iterate back to requirements specifications.

Section 3.1 explains how the inverse of a value function for device behavior is accomplished and why it is important. Section 3.2 discusses various aspects of events and what must be specified to complete a design. Section 3.3 makes a simplifying assumption about error and delay. However, if this assumption cannot be made, see Section 6.4 for a related discussion. Some features of CoRE that were not used in the case study are discussed in Section 3.4.

3.1 INVERTING IN AND OUT VALUE FUNCTIONS

The design approach discussed in this report involves estimating the value of environmental variables. The software estimates the monitored value from an input value, calculates an estimate of the desired controlled value, then determines an appropriate output value to achieve the desired behavior.

Figure 3 illustrates this flow of values through the software. IN' is the inversion of the IN value function, with monitored variables replaced by the corresponding approximations. For example, if the IN value function maps mon_Water_Temperature to in_Water_Temperature_Sensor, the IN' function will map in_Water_Temperature_Sensor to ~mon_Water_Temperature. The REQ' function is the REQ value function, with monitored and controlled variables replaced by their approximations. The OUT' function is the inversion of the OUT value function, with controlled variables replaced by their approximations.

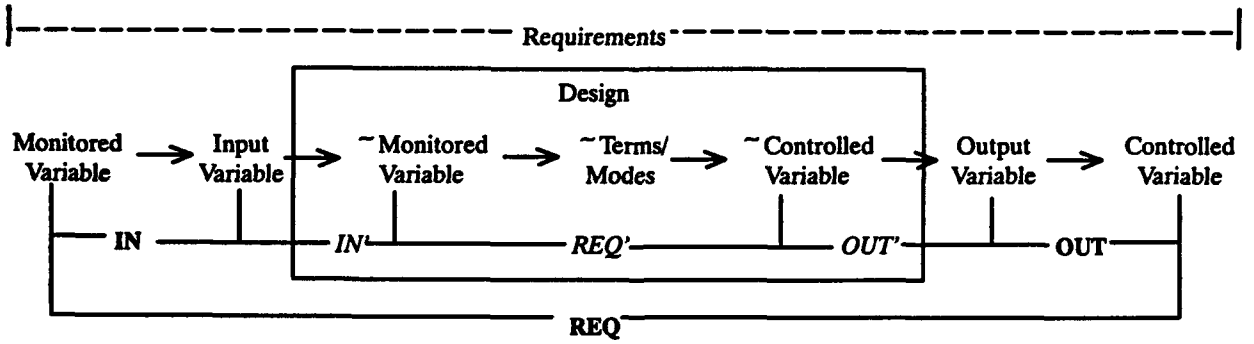


Figure 3. Flow of Values Through Software

The CoRE Guidebook mentions inverting a value function (see Section 11.3.3 of the CoRE Guidebook) but does not suggest that it is necessary. In software design, you will use the IN', REQ',

and OUT' functions rather than the value functions of IN, REQ, and OUT. **It does not matter whether you derive IN', REQ', and OUT' during requirements analysis or as the first activity in ADARTS design. What does matter is that you have them available for use in process structuring and class structuring.** If you derive these functions as part of design, you should do so before you begin process structuring or class structuring. Otherwise, you will have to derive them twice, resulting in unnecessary work and an increased probability of a mismatch between processes and objects. The mismatch would have to be resolved during software architecture design.

Table 3 contains an example of an IN value function and the corresponding IN' function in which IN describes a device that modifies two input variables: one to indicate the sign and the other, its unsigned value.

Table 3. Example of IN' Function

Function	Domain	Range	Definition
IN Value Function	mon_Temp	in_Sign, in_Value	in_Sign=sign(mon_Temp) in_Value=log(mon_Temp)
IN' Function	in_Sign, in_Value	~mon_Temp	~mon_Temp=in_Sign*10^{in_Value}

3.2 FREQUENCY OF EVENTS

To build an ADARTS design, a frequency profile must be specified for each CoRE event. The designer derives ADARTS external events, timer events, message communication, and process logic based on the expected throughput driven by event frequency and tolerable delay. Performance analysis is based on event frequency and tolerable delay.

You should explicitly state a frequency for each event before beginning process structuring. For periodic events, the frequency can be expressed in the event expression (@T(mon_Time mod 10)) or as part of an associated variable definition (see Section 4.2.1 of the CoRE Guidebook). For on-demand events (usually in an event table), the maximum frequency (minimum time between events) is required. Other characteristics, such as mean time between events, can also be helpful but are not required. The frequency of events related to the behavior of input and output devices (e.g., device interrupts) must also be specified.

The rest of this section explains why this frequency information is important by looking at how software gets the value of an input variable. Even if an event is not periodic, there is a window of opportunity for the software to use an input variable before it changes. This time interval is illustrated in Figure 4.

Stepping through the illustration chronologically: (1) an event occurs; (2) an initiation delay (optionally zero) is required before the input variable can change; (3) the input variable must change; (4) the window of opportunity begins for the software to use the variable; (5) the next event (for this event class) occurs; (6) the initiation delay expires, (7) ending the window of opportunity for using the value of the variable resulting from the first change.

If the software polls the input variable, this window of opportunity suggests a maximum interval between successive timer events to ensure that no change in value is missed. If the software responds to an interrupt, this window of opportunity suggests how quickly an interrupt must be processed.

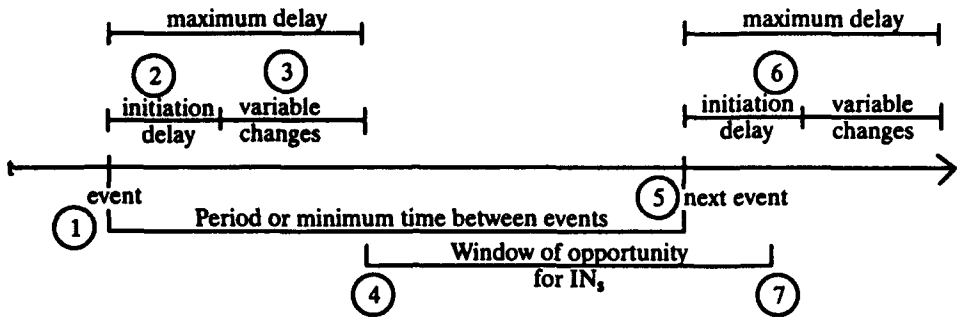


Figure 4. Illustration of Deriving Period or Maximum Delay for INs

Without knowing the minimum time between events, there is no way for the designer to allocate timing behavior to the IN_S process. Other constraints may require stricter constraints on IN_S, but the designer must still verify that IN_S acquires the input variable before it is lost.

3.3 ERROR AND DELAY

A key benefit of ADARTS is the separation of activities for designing the dynamic and static views of the software. If the design decisions in each activity depended on each other, this benefit would be essentially lost. Ideally, you will be able to deal with delay in process structuring and error in class structuring. This is possible if there is no dependency between error and delay or if the dependency is not strong. However, error and delay can be mutually dependent. When they are dependent, two possibilities are to:

- Make cursory analyses of error and delay during class structuring and process structuring, respectively. Conduct the complete analysis in software architecture design (see Section 6.4) and iterate to previous activities if necessary. This does leave the designer more flexibility in choosing a design along with the more complex analysis.

- Derive independent functions of error and delay. Most of this report assumes that the error and delay functions are independent, whether as specified by CoRE or derived before attempting an ADARTS design.

Figure 5 illustrates a simple dependence between error and delay. The line between points b and c represents the potential relationship between error and delay. Acceptable (REQ) or actual (IN/OUT/NAT) behavior lies below the diagonal line. To deal with error and delay independently during design, error and delay must be specified independently. In the case of REQ, values of error

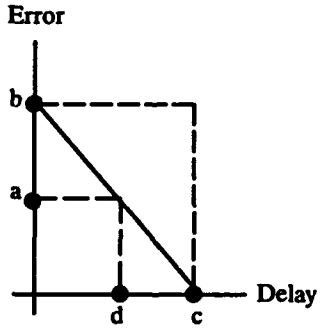


Figure 5. Illustrating Dependency Between Error and Delay

(a) and delay (d) must be selected so that if the software meets both tolerances independently, it meets the more lenient dependent tolerances. In the case of input devices (IN/OUT) and other behavior (NAT), worst case behavior must be assumed: maximum error (b) and maximum delay (c). Again, if the software meets timing constraints, assuming worst case behavior in both error and delay, it will meet the more lenient dependent tolerance.

3.4 CoRE REQUIREMENTS ARTIFACTS NOT USED

The case study does not use initiation and termination events (see Section 4.2.1 of the CoRE Guidebook) or sustaining conditions (see Section 4.3.1 of the CoRE Guidebook). This report does not make any recommendations on using these requirements artifacts during design.

4. PROCESS STRUCTURING

The ADARTS process structuring activity allows you to capture a dynamic software design that shows how processes interact to produce responses from stimuli. A process has its own thread of control that executes concurrently with other processes in the system. Concurrency and timing issues can be addressed in part by analysis of the system's processes and their interactions with each other and with the environment.

ADARTS software design heuristics, including process and class structuring, are based on the use of real-time structured analysis (RTSA) for specifying software requirements. This section explains how to derive an ADARTS process structure from a CoRE software requirements specification while limiting impact on the ADARTS process structuring activity as defined for RTSA. Additionally, this section describes how to maintain the higher degree of precision provided by CoRE by expressing process behaviors in terms of events to which a process must respond and the sequence of actions associated with each event (the use of this notation is optional).

When developing a process structure from an RTSA specification, the first step is to map elements of the RTSA specification to an "initial" (or "preliminary") process architecture. The initial process architecture is intended to provide the opportunity for maximum concurrency without regard for the inherent overhead, such as interprocess communication and context switching. The intent is to allow the designer to apply the ADARTS process clustering heuristics unmodified, regardless of the form in which the software requirements have been specified. The approach described in this report concentrates on this first activity—mapping elements of a CoRE specification to an initial process architecture. The initial process architecture provides a basis upon which ADARTS process clustering heuristics can be applied. During process clustering, you reduce the number of processes so that the advantages of concurrency, scheduling flexibility, and maintenance are balanced with performance requirements and complexity.

This section is not meant to replace the process structuring section (Section 8) of the ADARTS Guidebook, it is meant to supplement that section when you are using CoRE to specify software requirements. For example, this section does not describe the use of entity modeling during process structuring (Section 8.5 of the ADARTS Guidebook), which does not imply that entity modeling should not be used during process structuring because it is not part of CoRE.

When using a CoRE specification as the front end to ADARTS, you will derive the process architecture from the CoRE behavioral model, including:

- Variables (monitored, controlled, input, and output) and terms
- Relations (REQ, IN, OUT, NAT)

This section is divided into a number of subsections describing how the use of a CoRE specification affects an individual step of the ADARTS process structuring activity:

- Deriving an ADARTS initial process architecture from a CoRE specification (see Section 4.1)
- Developing process behavior specifications and maintaining the degree of precision provided by CoRE (see Section 4.2)
- Applying ADARTS process clustering criteria iteratively to consolidate processes (see Section 4.3)
- Identifying process communication and synchronization (see Section 4.4)
- Analyzing the design using the evaluation criteria (see Section 4.5)

Section 4.6 describes possible areas of future work.

4.1 DERIVING THE INITIAL PROCESS ARCHITECTURE

When performing the ADARTS process structuring activity, you first map requirements artifacts to an initial process architecture and then cluster (or combine) processes to reduce complexity and the overhead introduced by large numbers of concurrent processes. The ADARTS initial process architecture is a snapshot of the process architecture taken immediately after mapping from a requirements specification but before process clustering begins. This section describes how to map from the elements of a CoRE software requirements specification to the set of processes in the ADARTS initial process architecture. Use this section with Section 8.3 of the ADARTS Guidebook. The objectives of this mapping are as follows:

- To provide an initial process architecture that:
  - Isolates potentially concurrent activities
  - Captures finite state machines
  - Identifies the need for data storage
  - Captures the dynamic characteristics of hardware devices with which the software must interact
- To maintain the degree of precision provided by CoRE
- To allow the designer to apply the ADARTS process clustering heuristics unmodified (see Section 8.11 of the ADARTS Guidebook), regardless of the form in which the software requirements have been specified

In ADARTS, a process represents a sequential thread of execution that detects and reacts to a stimulus. Section 4.1.1 describes the basis of the initial mapping — how to identify these stimuli from events in a CoRE model. Events of interest for the initial mapping are related to the following CoRE artifacts:

- Input and output variables (see Section 4.1.2)
- Monitored and controlled variables (see Section 4.1.3)

- REQ value functions (see Section 4.1.4)
- Mode machines (see Section 4.1.5)
- Terms (see Section 4.1.6)

In addition, the initial process architecture must specify the need for data storage. Section 4.1.7 describes how to identify data stores.

Figure 6 illustrates the general scheme for the initial mapping, where parallelograms represent processes, arrows between processes represent data flow, and arrows attached to data stores represent recording or usage of approximations of monitored variables. Figure 6 illustrates the same flow of data as illustrated in Figure 3, except requirements have been allocated to processes. The initial process architecture indicates only data flow and data dependencies (e.g., IN_t processes depend on IN_s processes to supply the values of input variables), not necessarily message communications, which will be identified after process clustering.

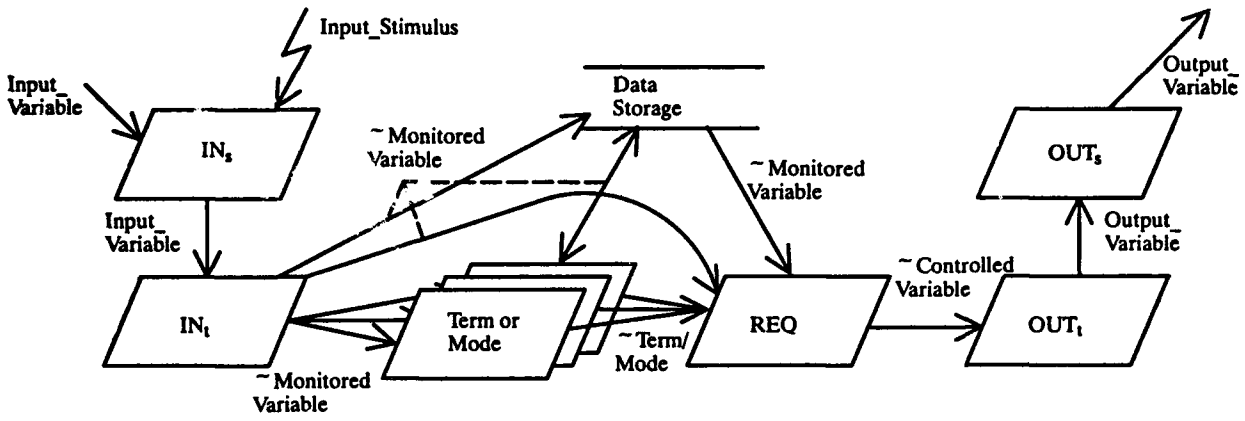


Figure 6. General Scheme for Initial Process Architecture

Table 4 characterizes the kinds of processes in the ADARTS initial process architecture. Section 4.1 describes the mapping in more detail.

Table 4. Characterization of Processes in Initial Mapping

Type of ADARTS Process	Purpose	Inputs	Outputs	Basis for Process Behavior
Input Variable (IN_s)	To acquire the value of an input variable from a device	Hardware interface with input device	Input variable	Frequency and device information related to input variable
Monitored Variable (IN_t)	To approximate a monitored variable from input variable(s)	Values of one or more input variables	Approximation of a monitored variable	Inversion of an IN value function

Table 4, continued

Type of ADARTS Process	Purpose	Inputs	Outputs	Basis for Process Behavior
Mode	To execute a finite state machine	Approximations of one or more monitored variables and/or terms	Current operating mode	Mode machine
Term	To calculate the value of a term based on the occurrence of an event	Approximations of one or more monitored variables and/or terms	Approximation of term	Term definition
REQ	To approximate the value of a controlled variable	Approximations of one or more monitored variables and/or terms	Approximation of a controlled variable	REQ' (the REQ value function expressed in terms of variable approximations, see Table 2)
Controlled Variable (OUT_t)	To calculate an output variable from the approximation of controlled variable(s)	Approximation of one or more controlled variables	Output variable	The inversion of an OUT value function
Output Variable (OUT_s)	To submit the value of an output variable to a device	Output variable	Hardware interface with output device	Frequency and device information related to output variable

Before mapping to an initial process architecture, you should specify the IN', OUT', and REQ' derived value functions as described in Section 2.6.3. The IN' derived value function identifies how the software can approximate the value of a monitored variable given the value of an input variable. The OUT' derived value function identifies how the software can set the value of a controlled variable using output variables. The REQ' derived value function describes the required behavior of the software in terms of approximating the value of a controlled variable given the approximations of one or more monitored variables or terms. Throughout this section, inverted IN and OUT value functions are referenced using the notation IN' and OUT' for the purpose of brevity.

4.1.1 Stimuli

In ADARTS, a process represents a sequential thread of execution that detects and reacts to a stimulus. Processes in the initial process architecture are derived from events described in the artifacts of a CoRE behavioral model. Stimuli in an ADARTS design are design artifacts that indicate the detection of CoRE events.

A periodic stimulus may appear in a number of different forms in an ADARTS design:

- A periodic event triggered internally by a timer, which may be implemented by hardware or software
- An event triggered by an external device that happens to occur on regular intervals
- A message passed between processes that happens to occur on regular intervals because the originating process responds to a periodic stimulus by passing a message

An aperiodic stimulus may appear in the following forms in an ADARTS design:

- An event triggered by an external device that does not occur on regular intervals
- A message passed between processes that does not occur on regular intervals because the originating process responds to an aperiodic stimulus by passing a message

In a CoRE specification, events are occurrences of a change in a conditional value. Events take the form of requirements that must be satisfied either periodically or upon demand (i.e., asynchronously). When analyzing events in a CoRE model to identify stimuli for ADARTS processes, be sure to only consider distinct events (see Section 2.6.2). Sets of events that are not distinct can be considered the same event for the purposes of ADARTS process structuring. Table 5 identifies possible sources of asynchronous and periodic events of interest to ADARTS in a CoRE behavioral model and includes examples of each.

Table 5. Identifying CoRE Events for ADARTS

CoRE Artifacts	Indicators of CoRE Events	
	Periodic	**Asynchronous**
Event tables defining value functions	@T(a function of time), e.g., @T[(mon_Time MOD 10 seconds) = 0]	@T(condition) or @F(condition), e.g., @F(in_Input_Variable = 0)
Timing requirements or timing behavior associated with controlled variables (or their corresponding REQ relations)	Periodic scheduling constraints imply periodic events of a given frequency (typically indicated by condition or selector tables).	Demand scheduling constraints indicate asynchronous events that should be identified in the REQ relation (defined by an event table), e.g., @T(mon_Monitored_Variable = 0)
Device information associated with input or output variables (or their corresponding IN and OUT relations)	Devices that periodically produce software inputs or consume software outputs	Active devices (i.e., asynchronous, interrupt-driven devices)
	Passive (i.e., continuous) devices may indicate either periodic or asynchronous events, depending upon the software's need to retrieve inputs or produce outputs.	

The behavior of the REQ process may be used to drive the events and message decisions outward toward the IN_s and OUT_s processes that interact with devices (see Figure 6). An event in REQ' may occur upon a change in the approximation of a monitored variable. But for the purpose of reducing

communication, the logic of the IN_s and IN_t processes involved in the approximation of the monitored variable may be modified to limit the frequency with which they respond. For example, if an IN_s process periodically samples an input variable and passes its value to an IN_t process, then the IN_t process need not react unless it detects a change in the approximation of the monitored variable. Equivalently, the sampling rate of the IN_s process may be decreased such that it detects changes in the values of input variables more efficiently (i.e., its period may be reduced based on known characteristics of a monitored variable, such as those defined in NAT relations). The logic of the REQ process need not be concerned about detecting a change in the approximation of the monitored variable; this occurrence is assumed upon receipt of the value of an approximation of a monitored variable.

Events in a CoRE behavioral model are used to identify stimuli in an ADARTS initial process architecture and, therefore, the processes that respond to stimuli. Sections 4.1.2 through 4.1.6 describe how to generate an initial process architecture from the artifacts of a CoRE model according to the indicators described in Table 5.

4.1.2 Input and Output Variables

The mapping of CoRE input and output variables to processes in the initial process architecture is based on the need to retrieve or produce data. The guidelines in this section are similar to the guidelines in the ADARTS Guidebook for identifying processes that interact with devices (see Section 8.4 of the ADARTS Guidebook).

Processes that retrieve input variables are called IN_s processes. Processes that produce output variables are called OUT_s processes. The work performed by an IN_s or OUT_s process is to respond to a stimulus indicating the need to retrieve an input or produce an output. Figure 7 shows an example of an IN_s process that responds to a device interrupt by sampling the value of an input variable and forwarding it. Figure 8 shows an example of an OUT_s process that responds to the receipt of an updated output variable by submitting that variable to an output device, which uses the output variable to modify a controlled variable.

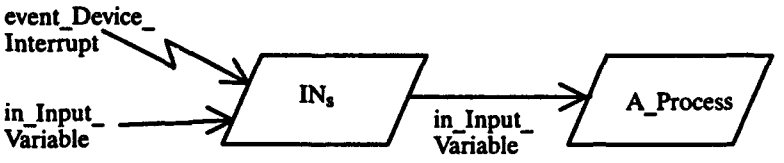


Figure 7. IN_s Process Example

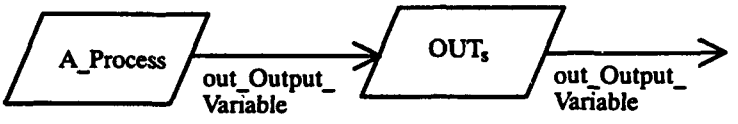


Figure 8. OUT_s Process Example

For each CoRE input and output variable, begin by creating a process whose job is to react to a stimulus by acquiring the value of an input variable from a device or submitting the value of an output variable to a device. There are three kinds of devices that determine the stimuli that cause IN_s and OUT_s processes to respond:

- Active devices (see Section 4.1.2.1) signal interrupts when input variables have been updated or when output variables should be produced by the software.
- Periodic devices (see Section 4.1.2.2) produce input variables or consume output variables at regular intervals.
- Passive (or continuous) devices (see Section 4.1.2.3) produce input variables or consume output variables transparently to the software (typically at very high periodic frequencies).

Input and output variable definitions should indicate whether input and output devices are active, periodic, or passive. Use corresponding input variable definitions to identify the stimulus causing an IN_s process to respond. Equivalently, use corresponding output variable definitions to identify the stimulus causing an OUT_s process to respond. For a process that interacts with an active device, the stimulus is an external event (i.e., a device interrupt); for a process that interacts with a periodic device, the stimulus is a periodic event; and for a process that interacts with a passive device, the stimulus will be identified later, based on the corresponding REQ value function (see Section 4.1.4).

Typically, each input variable and output variable maps to a single process that retrieves input or produces output in response to receipt of a message or occurrence of a timer or external event (see Section 4.1.1). Less frequently, an input variable or output variable will map to multiple processes, e.g., one executing periodically and another executing upon demand (i.e., asynchronously). Figure 9 shows an example of two IN_s processes that react to different stimuli but sample the same input variable. The process IN_s_Periodic samples in_Input_Variable upon occurrence of the event event_Periodic_10_Second. The process IN_s_Demand process samples in_Input_Variable upon receipt of Need_to_Sample_Input.

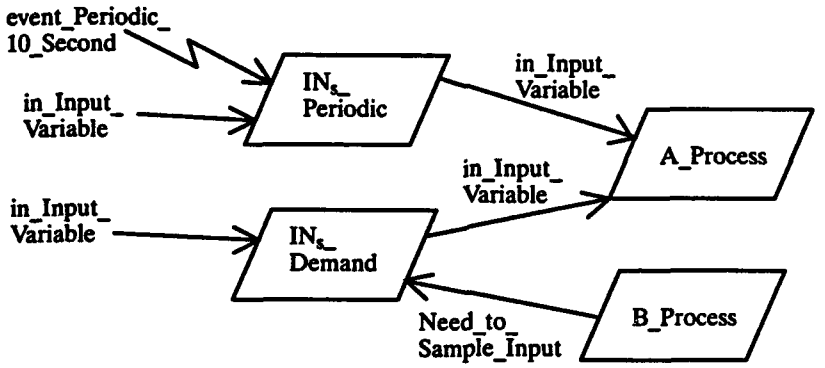


Figure 9. Periodic and Demand IN_s Processes Example

4.1.2.1 Active Devices

If a device signals an interrupt indicating the availability of an input variable (or the need to produce an output variable), use an external event to activate the IN_s (or OUT_s) process. Input and output variable definitions should indicate the interrupt(s) signaled by active devices. Event/response pairs in an event table defining the corresponding IN' or OUT' value function should indicate the required responses to device interrupts. For example, Figure 10 illustrates the IN' value function corresponding to the active emergency button device in the Host-at-Sea (HAS) Buoy case study in the Appendix.

The events in Figure 10 are defined as follows:

Event	~mon_Emergency_Button
event_Button_Indicator_Set	"Pressed"
event_Button_Indicator_Reset	"Released"

Figure 10. IN' for mon_Emergency_Button

```
event_Button_Indicator_Set = @T(in_Button_Indicator = "2#1xxxxxxx#")

event_Button_Indicator_Reset = @T(in_Button_Indicator = "2#0xxxxxxx#")
```

Figure 11 illustrates the resulting IN_s process activated by the interrupt from the emergency button. The process Monitor_Button_Indicator interfaces with the emergency button device by detecting the interrupt Button_Interrupt, sampling the input variable Button_Indicator, and passing its value to another process. In this case, you are really only concerned with event_Button_Indicator_Set because there is no need to respond to the button being released. Therefore, the process logic could be simplified and the amount of message communication could be reduced by ignoring the event indicating that the button has been released.

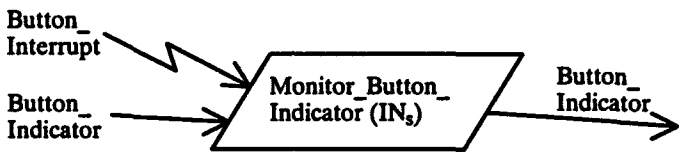


Figure 11. IN_s Process Activated by a Device Interrupt

4.1.2.2 Periodic Devices

If a device periodically updates an input variable or periodically reads an output variable, use a timer event with the same period as specified by the CoRE description of the device behavior. Later, when fine-tuning the process architecture, you should consider whether all samples of the input and output variables are of interest to the software. If not, you may decide to change the IN_s or OUT_s process to a demand-driven process or to reduce the frequency of its activation.

For example, Figure 12 illustrates the IN' value function corresponding to the Omega navigation system in the HAS Buoy case study in the Appendix.

Event	~mon_Buoy_Location
event_Periodic_ 30_Second	Latitude <= (Degrees <= MAX(<Latitude>in_Omega_System_Input.Bytes_1&2, 359), Minutes <= MAX(<Latitude>in_Omega_System_Input.Byte_3, 59), Seconds <= MAX(<Latitude>in_Omega_System_Input.Byte_4, 59) + MAX(<Latitude>in_Omega_System_Input.Byte_5, 99) / 100), Longitude <= (Degrees <= MAX(<Longitude>in_Omega_System_Input.Bytes_1&2, 359), Minutes <= MAX(<Longitude>in_Omega_System_Input.Byte_3, 59), Seconds <= MAX(<Longitude>in_Omega_System_Input.Byte_4, 59) + MAX(<Longitude>in_Omega_System_Input.Byte_5, 99) / 100)

Figure 12. IN' for mon_Buoy_Location

The event in Figure 12, defined as follows, indicates that the approximation of the monitored variable could change value periodically, at 30-second intervals, based on the update rate of the input variable in_Omega_System_Input:

```
event_Periodic_30_Second = @T([mon_Time MOD 30 seconds] = 0)
```

Figure 13 illustrates a periodic IN_s process activated at the same frequency with which the input device updates the value of its corresponding input variable. The process Monitor_Omega_System_Input interfaces with the Omega system by sampling the input variable Omega_System_Input periodically upon occurrence of the timer event Time_30. The value of Omega_System_Input is passed on to another process.

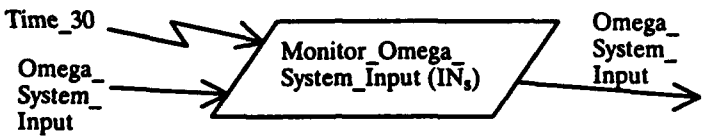


Figure 13. IN_s Process Activated Periodically

You should try to synchronize the periodic intervals of IN_s processes and input devices to minimize the delay in which the software recognizes changes in the values of input variables. Equivalently, you should try to synchronize the periodic intervals of OUT_s processes and output devices. Without any attempt at synchronization, the worst case scenario introduces a delay equal to the period.

4.1.2.3 Passive Devices

If a variable is produced or consumed continuously by a device (i.e., the device is passive) or upon detection of a software-driven interrupt, the stimulus must be determined from REQ'. For an input variable, identify the expressions in REQ' involving approximations of monitored variables calculated from the input variable under consideration. For an output variable, identify the expressions in REQ' involving approximations of controlled variables that affect the values of the output variable under consideration.

For each event associated with the expressions, use the frequency profiles for the corresponding events in the CoRE specification and allotted time for the value function to be evaluated to determine how often the input variable must be sampled or the output variable must be produced. As a result, you will either:

- Use a periodic stimulus causing the input variable to be polled or output variable to be produced at regular intervals.
- Use the receipt of a message from another process (as described in Section 4.1.4) if an input variable must be retrieved or output variable must be produced upon demand by another part of the software system.

Determining the ideal frequencies of periodic IN_s and OUT_s processes that interface with passive devices from a REQ value function is nontrivial. There are a number of requirements artifacts that affect the frequency with which IN_s processes (and OUT_s processes) should be activated:

- The frequency with which a device updates the value of an input variable and the tolerable delay specified by relevant REQ relations. For example, assume a temperature sensor

measures air temperature every 10 seconds, with the first measurement taken at time t_0 (i.e., the intervals of the process and the device are in phase). The periodic frequency of the corresponding IN_s process and its synchronization with the device have an effect on the average and worst case delay in the software recognizing updates to the input variable:

  - If the device and the IN_s process are in phase (i.e., the IN_s process takes its first sample at or immediately after time t_0), a period of 10 seconds will provide the best average and worst case delays possible. Increasing the IN_s process' sampling rate will not reduce delay and could even increase it.

  - If the device and the IN_s process are not in phase, average and worst case delays are functions of the period of the IN_s process and the difference between t_0 and time that IN_s takes its first sample. In this case, increasing the sampling rate of the IN_s process will improve average and worst case delay, and a period of 10 seconds will provide a maximum delay of 10 seconds.

- NAT relations that specify the maximum rate of change of the monitored variable are measured by an input variable. You may be able to determine the maximum necessary sampling rate of the IN_s process from the maximum rate of change of a monitored variable and the accuracy requirements for controlled variables affected by changes in a monitored variable. For example, assume that an accuracy requirement for reporting speed is $\pm$ 0.5 mph and there is a NAT relation stating that $|\Delta \text{ mph} / \Delta t| \leq 10$ mph/s. Therefore, it takes at least 0.5/10 = 0.05 s for the car to change speed by 0.5 mph. So, the periodic sampling rate need not be greater than 20 Hz in a perfect world. However, there is delay introduced by the software and the devices it uses that must be factored into the sampling rate.

In the HAS Buoy case study, the REQ relation for the controlled variable con_Report (see Section App.2.11.1) indicates that wind and temperature reports must be produced every 60 seconds. Wind and temperature reports contain water temperature readings that must be accurate within a certain degree. Assuming that six samples of water temperature readings per minute are required to obtain the required accuracy (as was assumed in the case study), use a periodic IN_s process activated every 10 seconds (similar to the IN_s process in Figure 13).

It is possible for both periodic and asynchronous stimuli to activate the IN_s or OUT_s processes that interface with passive devices when the variable is indirectly involved in multiple REQ relations. In this case, create two processes: one activated by a periodic event and one activated asynchronously (as illustrated in Figure 9). It is also possible to determine the need from different REQ relations to sample an input variable or to produce an output variable at different periodic intervals. In this case, you may be able to use a single periodic stimulus occurring at the higher frequency.

4.1.3 Monitored and Controlled Variables

For each monitored variable in a CoRE model, use the inverse of the IN value function, IN', to define an IN_t process for each unique event (see Section 2.6.2) used in the function. Each IN_t process reacts to a stimulus by changing the value of the approximation of a monitored variable. Use the IN' value function to specify the approximation of a monitored variable from input variable(s) performed by each IN_t process.

The inputs to an IN_t process are one or more input variables; the output is an approximation of a monitored variable. Figure 14 shows an example of an IN_t process that responds to the receipt of either

of two different input variables because two input variables are used in approximating a monitored variable.

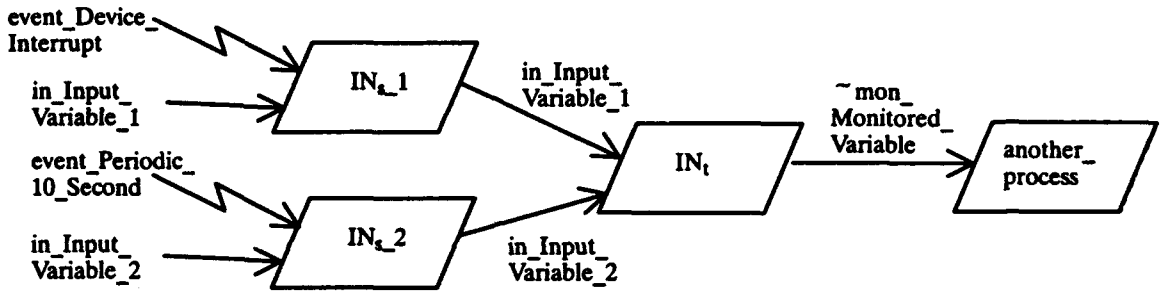


Figure 14. IN_t Process Example

If time (e.g., "mon_Time") is a monitored variable in your CoRE behavioral model, you may choose not to create an IN_t process if you intend to use your run-time system to determine "current time" and to implement periodic behavior. Examine the physical description of the monitored variable to make your decision.

For each controlled variable in a CoRE behavioral model, use the inverse of the OUT value function, OUT', to define an OUT_t process for each unique event used in the function. Each OUT_t process reacts to a stimulus by determining what the value of an output variable should be. Use the OUT' value function to specify the calculation of an output variable from controlled variable approximation(s).

The inputs to each OUT_t process are approximations of one or more controlled variables; the output is an output variable. Figure 15 shows an example where two OUT_t processes are necessary because two output variables are required to set the value of the controlled variable.

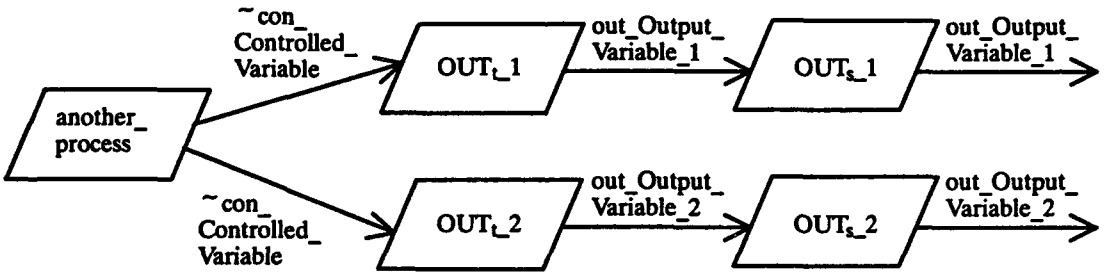


Figure 15. OUT_t Process Example

It may be tempting to create a single process that performs the work of both the IN_s and IN_t processes, i.e., one that responds to a stimulus indicating the need to retrieve an input variable and translate it into the approximation of a monitored variable (i.e., a single IN process instead of separate IN_s and IN_t processes). Equivalently, the same temptation may exist to use a single process for translating an approximation of a controlled variable into an output variable and sending the output variable to a device (i.e., a single OUT process instead of separate OUT_s and OUT_t processes). The rationale for maintaining this separation is illustrated in Figure 16, where the shaded ovals represent possible temporal cohesion and the unshaded ovals represent sequential cohesion. It may seem unnecessary to separate the IN_s_1 and IN_t_1 processes in Figure 16 because of the obvious sequential cohesion. However, there may also be temporal cohesion between pairs of processes, such as IN_s_1 and IN_s_2,

for which a greater benefit is obtained by clustering. Therefore, by adhering to the recommended mapping, you allow more flexibility when making tradeoffs during process clustering (see Section 4.3).

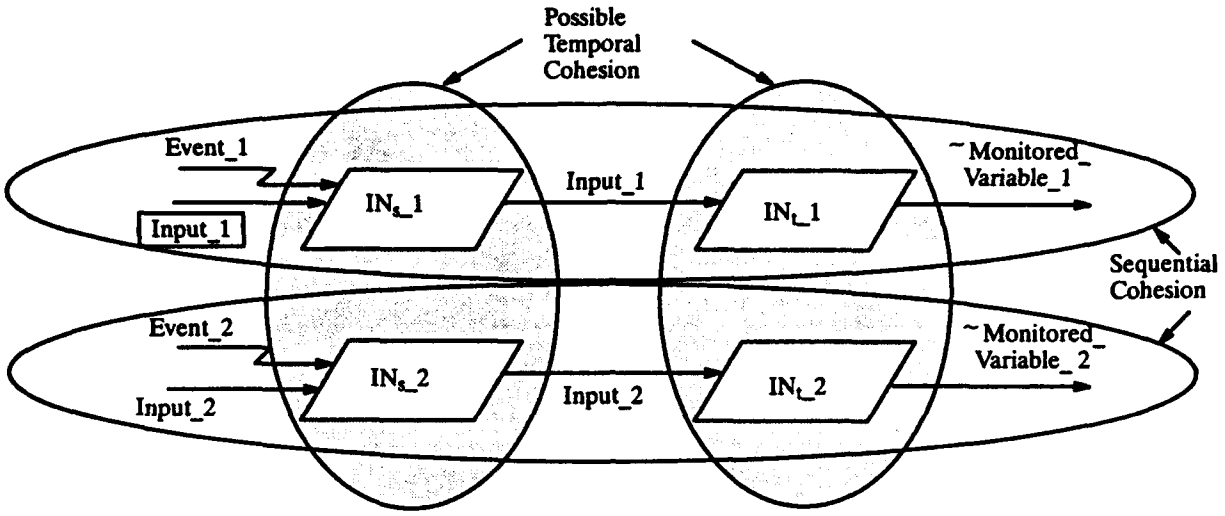


Figure 16. Rationale for Mapping to Initial Process Architecture

In Figure 16, temporal cohesion exists between IN_s_1 and IN_s_2 if Event_1 = Event_2 or may exist if both events are periodic. Temporal cohesion may also exist between IN_t_1 and IN_t_2 if temporal cohesion exists between IN_s_1 and IN_s_2. Sequential cohesion always exists between the pairs (IN_s_1, IN_t_1) and (IN_s_2 and IN_t_2).

An example of when it is beneficial to maintain the separation of IN_s and IN_t processes is when the approximation of a controlled variable performed by the IN_t process is time consuming and the IN_s process must be able to handle bursts of interrupts. In this case, the IN_s process is allowed to handle the bursts of interrupts and the IN_t process can perform calculations when the activity of the IN_s process has slowed down.

4.1.4 REQ Value Functions

Sections 4.1.2 and 4.1.3 described how to identify IN_s, IN_t, OUT_s, and OUT_t processes. This set of processes is roughly the equivalent of the set of device interface processes described by the ADARTS Guidebook. What remain to be identified are the internal environment-independent processes. The shaded area in Figure 17 indicates the kinds of processes (including data stores) from the general scheme that remain to be identified. This section describes how to identify the REQ processes.

REQ relations are specified in CoRE using event, condition, or selector tables. Map a condition table or selector table to a single process activated periodically. For event tables, map each unique event (see Section 2.6.2) to a process.

Each of these REQ processes represents a potentially concurrent transformation from approximations of monitored variables to an approximation of a controlled variable. Section 4.1.4.1 describes how to derive processes in the initial process architecture from REQ value functions defined by event tables. Section 4.1.4.2 describes how to derive processes from REQ value functions defined by condition and selector tables.

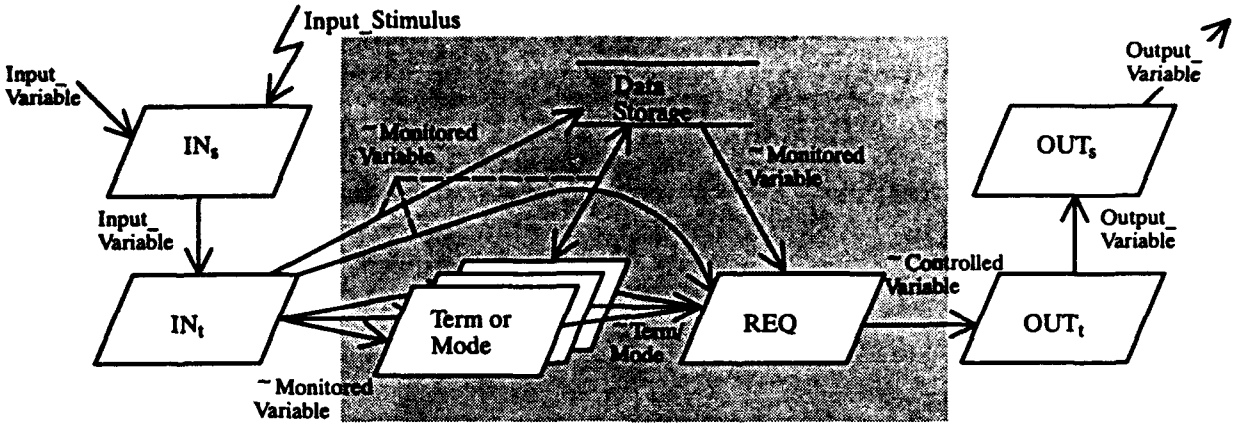


Figure 17. General Scheme for Initial Process Architecture

4.1.4.1 Event Tables

Processes are derived from REQ value functions described in the form of event tables based on the need to respond to distinct, potentially concurrent events. If the same event is identified more than once in the REQ value function (i.e., they represent the same event occurring under different circumstances, such as in different modes), use a single process to respond to all occurrences of the event. For example, @T(mon $\leq$ 50) and @F(mon > 50) are the same event represented differently. On the other hand, @T(mon > 0) and @T(mon $\leq$ 10) are separate, distinct events (as are periodic events with different periods or periodic events with the same period that are not in phase).

From the set of distinct events in an event table, identify those to which the software may be required to respond concurrently (i.e., after one event occurs, the second event occurs, and the software must be able to respond to the second event, even if it has not yet completed its response to the first event). Consider multiple events to which the software can only respond sequentially as a single stimulus and, therefore, a single process. For example, if two events are defined by a Boolean variable taking on the values "True" and "False," the responses to these events are not likely to be performed in parallel because a single instance of the Boolean variable is either "True" or "False" at any given point in time. Also, events defined conditionally, such as "@T(C_1) when C_2" and "@T(C_1) when C_3" represent one distinct event "@T(C_1)" that can be handled by actions taken conditionally upon detection of the event.

For each REQ value function defined by an event table, create one process for each of these distinct, potentially concurrent events. These processes are called REQ processes. Each resulting REQ process describes the actions performed in response to a distinct event and approximates the value of a controlled variable from one or more approximations of monitored variables and terms and modes. Then determine from REQ' how each process approximates the value of a controlled variable.

In the HAS Buoy case study, there are two REQ relations that exemplify the derivation of processes from event tables. In the REQ' function for the approximation of con_Red_Light, there are two different events: event_Red_Light_On and event_Red_Light_Off (see Figure 18), defined as follows:

```
event_Red_Light_On = @T(~mon_Light_Command = "Red_Light_On")

event_Red_Light_Off = @T(~mon_Light_Command = "Red_Light_Off")
```

There is no need to simultaneously turn the light on and off (because there is only one light); therefore, only one process is necessary in the initial process architecture to handle both events. As illustrated in Figure 19, the events are detected by receipt of Light_Command, and the need to turn the light on or off is determined by looking at the value of Light_Command (either "On" or "Off").

Event	~con_Red_Light
event_Red_Light_On	"On"
event_Red_Light_Off	"Off"

Figure 18. REQ' Function for con_Red_Light

Figure 19. REQ Process Process_Red_Light_Request

As another example, the REQ' function for the approximation of con_Report contains five events: event_Periodic_60_Second (occurs twice), event_Airplane_Detailed_Report_Request, event_Ship_ Detailed_Report_Request, and event_History_Report_Request, as illustrated in the Appendix. In this case, four processes were created (see Figure 20):

- ***Generate_Periodic_Reports***: Responds to both of the periodic events because:
  - They are the same event.
  - The responses cannot be carried out in parallel because the prerequisite for each of the events is a particular mode (i.e., "mod_SOS" or "mode_Normal") and the software is only in one of the two modes at any given time.
- ***Generate_History_Report, Generate_Ship_Detailed_Report, and Generate_Airplane_Detailed_Report***: Each of which responds to a particular kind of Vessel_Request and may execute in parallel.

4.1.4.2 Condition and Selector Tables

REQ value functions defined by condition and selector tables typically represent periodic behavior and should map to a single process that responds to periodic events. The resulting process calculates the approximation of a controlled variable periodically. The frequency of activation for these processes is derived from the frequency of the periodic event associated with the table.

Table 6 illustrates an example of a condition table defining a REQ' function. Figure 21 illustrates how the process derived from it may appear on the initial process architecture diagram, assuming a requirement to modify the controlled variable every 10 seconds.

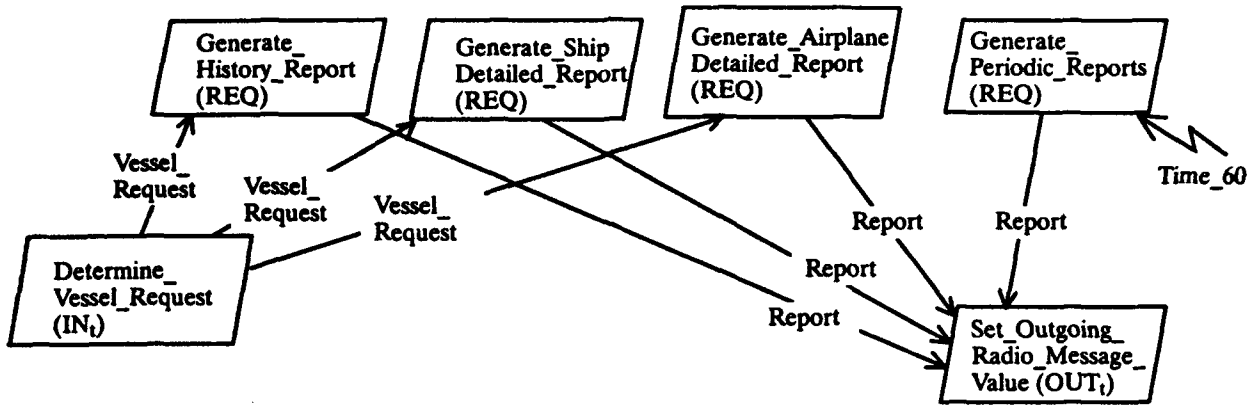


Figure 20. REQ Processes Supporting REQ_Relation_for_con_Report

Table 6. Deriving Processes From Condition Tables

Mode	Condition	
mode_Normal	~mon_Temperature < 100°C	~mon_Temperature ≥ 100°C
mode_Degraded	~mon_Temperature < 90°C	~mon_Temperature ≥ 90°C
~con_Status_Light=	"Green"	"Red"

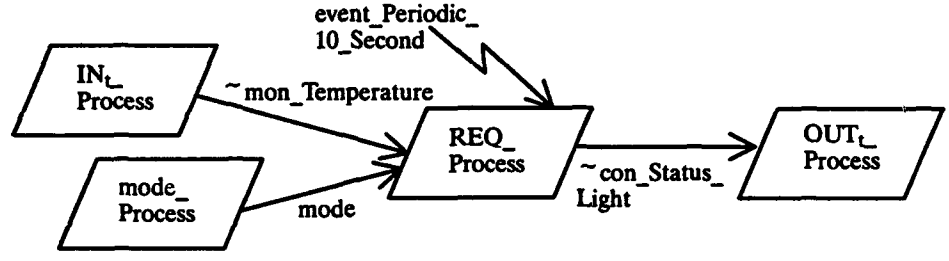


Figure 21. Periodic REQ Process

4.1.5 Mode Machines

Create a process for each mode machine in the CoRE specification. These processes are called mode processes, and their purpose is to track the current operating mode of the software and update it in response to events as stipulated by the mode machine.

Figure 22 illustrates the mode machine from the HAS Buoy case study. Figure 23 illustrates the process derived from the mode machine: it responds to the receipt of either of two kinds of messages that may cause the system to change state. It passes state change information to Generate_Periodic_Reports, the only process that is affected by state changes.

4.1.6 Terms

Create a process for each term that is defined using an event. These processes are called term processes. When the event occurs, the process calculates the value of the term, given the values of one or more approximations of monitored variables or other terms. Do not create a process for a term whose value is not calculated upon occurrence of an event.

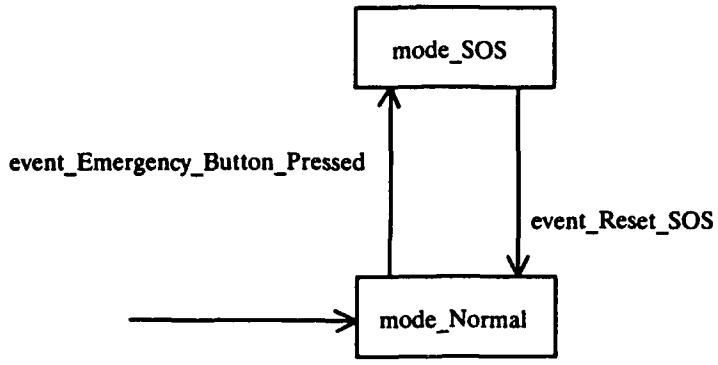


Figure 22. HAS Buoy Mode Machine

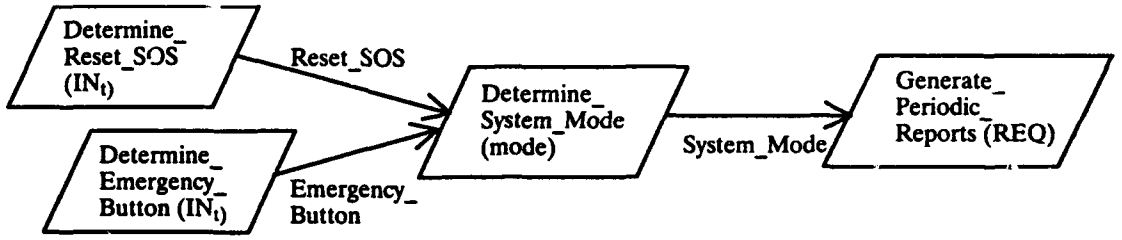


Figure 23. HAS Buoy Mode Process

Table 7 shows an example of a term defined by an event that implies the need for a process on the initial process architecture diagram. Table 8 shows an example of a term defined using a condition table that does not imply the need for a process.

Table 7. Term Defined by an Event

Event	term_Average_Temperature
@T(mon_Average_Needed = "True")	[mon_Temperature(t) + mon_Temperature(t−30)] / 2

Table 8. Term Defined by Conditions

Condition	term_Temperature_In_Range
0°C ≤ mon_Temperature ≤ 100°C	"True"
(0°C > mon_Temperature) OR (mon_Temperature > 100°C)	"False"

4.1.7 Determining the Need for Internal Data Storage

From the CoRE perspective, the need for internal data storage is a derived requirement identified during software design. The general rule of thumb for identifying the need for data storage from CoRE specifications is to look for references to the past found in value functions (delay terms are ignored). Search the REQ, IN, and OUT value functions, variables, and terms for references to past values of one or more monitored variables or terms. You should identify the content of each data store and the number of copies of data that it contains. The data store contains the corresponding approximations to monitored variables or terms.

For example, the following terms from the HAS Buoy case study indicate the need for internal data storage. Specifically, a data store containing six copies of mon_Air_Temperature data and a data store containing 2,880 copies of term_Wind_and_Temperature_Report are needed.

```
term_Averaged_Air_Temperature =
   ROUND [(SUM i: 0 <= i <= 5 : mon_Air_Temperature (t - 10 x i)) / 6]

term_Weather_History_Report =
   * The set of term_Wind_and_Temperature_Report(i), where i = t-136_800,
    t-136_740, ..., t (i.e., step by 60 seconds). That is, the
    term_Wind_and_Temperature_Report at every 60 second interval over the
    last 48 hours. *
```

4.2 SPECIFYING PROCESS BEHAVIOR

This section describes how to create process behavior specifications for processes in an ADARTS initial process architecture derived from a CoRE software requirements specification. Section 8.13.2 of the ADARTS Guidebook describes how to describe processes using process behavior specifications. This section only provides guidance in developing those parts of the specification that are affected by the use of CoRE for requirements analysis. The guidelines in this section are optional, facilitated by the precision of CoRE and motivated by the goal of maintaining CoRE's level of precision throughout design. You do not have to follow the guidelines in this section to produce an ADARTS design from CoRE requirements. However, your design will benefit from precision if you do follow these guidelines. The benefits of precision are described in Section 2.5.

In particular, the following parts of process behavior specifications are affected by the use of CoRE:

- Process logic (see Section 4.2.1)
- Process interfaces (see Section 4.2.2)
- Requirements traceability (see Section 4.2.3)

4.2.1 Process Logic

This section describes a form of process logic: one that uses an abstract stimulus/response notation describing process behavior on a thread-by-thread basis. The use of this form of process logic is not required to build an ADARTS design from a CoRE requirements specification, but it is recommended because it allows you to maintain CoRE's precision in the process structure in an understandable and unambiguous manner.

Section 4.2.1.1 introduces the stimulus-response notation. Section 4.2.1.2 describes an example of the use of the stimulus/response notation. Section 4.2.1.3 provides some rationale for the use of this notation.

4.2.1.1 Stimulus-Response Notation

Each process in the initial process architecture performs work when it responds to a stimulus. When specifying process logic, create stimulus/response pairs, each of which defines the work performed by

a particular process in response to a stimulus. By applying the guidelines in Section 4.1, you identify processes and the stimuli that cause them to respond from the artifacts of a CoRE model. Table 9 identifies the stimuli that may cause each kind of process in the initial process architecture to respond.

Table 9. Process Stimuli

Process	Relevant Artifacts	Stimuli
IN_s	input variable definition, IN' value function	External event (device interrupt), periodic event, or receipt of a message from another process
IN_t	IN'	Receipt of message from an IN_s process
Mode	Mode machine	Periodic event or receipt of message from term or IN_t process
Term	Term definitions	Periodic event or receipt of message from mode or IN_t process
REQ	REQ'	Periodic event or receipt of message from mode, term, or IN_t process
OUT_t	OUT'	Periodic event or receipt of message from REQ process
OUT_s	output variable definition, OUT'	External event (device interrupt), periodic event, or receipt of message from OUT_t process

When specifying the responses to stimuli, be sure to use any NAT relations that are relevant to the behavior you are specifying. The process behavior you describe should assume that every NAT relation holds true — there is no need to attempt to detect and react to violations of NAT relations. For example, in the HAS Buoy case study in the Appendix, there is a NAT relation defining the bounds of the monitored variable mon_Water_Temperature as follows:

```
-4 <= mon_Water_Temperature <= 100 (degrees Celsius)
```

Therefore, the IN_t process that translates the value of in_Water_Temperature_Sensor into the approximation of mon_Water_Temperature need not be concerned with values out of the specified range.

The response portion of stimulus/response pairs is an ordered set of actions that describes when the process interacts with devices, data stores, or other processes. The response must indicate the resources, including process inputs and stored information, required to produce outputs. In general, avoid describing the details of computations in process logic; you will encapsulate them in classes rather than make them explicit in process behavior specifications. However, you should make clear the dependencies between process inputs and outputs and computations performed by a process. That is, be sure to identify:

- The process inputs required to perform each computation
- Which computation results are required to produce each process output

Recording this information helps to identify candidates for process clustering, evaluate expected performance, and identify deadlock and race conditions.

The stimulus-response notation is an unordered list of stimulus/response pairs in the following form:

Stimulus: S_1 **when** C_1
S_2 **when** C_2
...
S_n **when** C_n
Response: A_1
A_2
...
A_m

Each S_i denotes a stimulus, whether it is an external or timer event or the receipt of a message. Each C_j, if present, identifies a qualifying condition under which the stimulus may be recognized. A response is specified as an ordered set of actions, with each action denoted by A_i. The order of a set of actions is significant in that it indicates the required order in which the actions must be performed unless otherwise indicated. An action can be a computation, access to stored information, generation of a message or external event, etc. If multiple stimuli appear in a single stimulus/response pair, the action(s) will be taken upon recognition of *any* of the stimuli. It is not mandatory that a process provide an externally visible response to each stimulus. For example, a stimulus may do no more than cause a process to store some data locally.

The set of stimuli (S_i) and actions (A_i) should indicate all externally visible activity of a process — evaluating a condition associated with an event should not require any externally visible activity (e.g., examination of data modified by other processes or interaction with other processes or the external environment). If you omit a condition, it is assumed to be **true** (i.e., the stimulus is always recognized and the actions always taken). If a stimulus occurs in a situation that satisfies the associated conditions in two stimulus/response pairs, assume that only one of the action sequences (selected nondeterministically[2]) will be executed. Selection of a single response is necessary because two or more responses may interfere with each other. Nondeterministic choice simplifies the notation by disregarding the order of stimulus/response pairs. It has the additional advantage of not overly constraining the implementor.

4.2.1.2 Process Logic Example

This notation describes process logic in terms of stimulus/response pairs that are similar in nature to the precondition/postcondition pairs sometimes used to describe serial computations. This notation is illustrated by applying it to the Generate_Periodic_Reports process of the HAS Buoy case study. Every 60 seconds, this process generates a message representing a report that is passed to another process for radio transmission. The generated message depends on the current operating mode of the buoy:

- If the buoy is operating in "SOS" mode, the message contains an SOS signal and the current buoy location.
- If the buoy is in "Normal" mode, the message contains weather information previously obtained from external sensors and recorded in data stores by other processes.

In addition, this process reacts to the events that cause a mode change. Specifically:

2. "Nondeterministic" is not the same as "random." "Random" choice of several possibilities means that each possibility has roughly the same chance of being chosen. "Nondeterministic" choice means that the designer does not care. The programmer (or run-time system) may make the choice in whatever way it pleases. Nondeterministic choice may be random, but it does not have to be.

- If the buoy is in "SOS" mode and a radio message requesting termination of transmission of SOS signals is received, the current mode is changed to "mode_Normal."
- If the buoy is in "Normal" mode and the emergency button is pressed, the current mode is changed to "SOS."

A summary of the stimulus/response specification for this process follows:

Stimulus: Time_60 (a timer event with a period of 60 seconds)
Response: If the buoy is in SOS mode, format and transmit an SOS message.
Otherwise, retrieve stored information about air temperature, water temperature, wind direction, and wind magnitude (i.e., speed), format this information into a Wind_and_Temperature_Report, and cause the report to be queued and transmitted.

Stimulus: Received a Mode_Change message
Response: If the buoy is in Normal mode and the Mode_Change message indicates that the Emergency Button was pressed, change the mode to SOS.
Otherwise, if the buoy is in SOS mode and the message indicates that the Reset_SOS radio message was received, then change the mode to Normal.

Section App.3.4.4 contains the detailed specification for the Generate_Periodic_Reports process.

4.2.1.3 Rationale

The stimulus/response notation is abstract and amenable to certain kinds of analysis (see Section 4.5). The benefit of abstraction is that it discourages inclusion of irrelevant detail in the process logic. Coding details do not belong in process logic specifications because they distract the designer from important design issues and constrain the programmer unnecessarily. You should make an effort to exclude coding details from design specifications just as you try to exclude design information from requirements specifications.

4.2.2 Process Interfaces

Identifying process interfaces for processes derived from an RTSA specification is straightforward: you map them from data flows and control flows between transformations. When the initial process architecture is derived from a CoRE specification, the mapping is not so straightforward. Typically, a series of processes from the initial process architecture derived from a CoRE specification will take the following form (as described in Section 4.1 and illustrated in Figure 6):

- External events (device interrupts or timer events) or messages from other processes cause an IN_s processes to sample the value of an input variable.
- Input variables are communicated via messages from IN_s processes to one or more IN_t processes.
- Approximations of monitored variables are communicated via messages from IN_t processes to term, mode, and/or REQ processes. Term processes produce terms and mode processes produce modes that are passed on to REQ processes via messages.
- Approximations of the ideal values of controlled variables are communicated via messages from REQ processes to OUT_t processes.

- Output variables are communicated via messages from OUT_t processes to OUT_s processes.
- Output variables are passed from OUT_s processes to output devices. OUT_s processes are typically activated by incoming output variables in the form of messages, device interrupts, or timer events.

Table 9 identifies the sources of relevant information in a CoRE model for each kind of process. Be sure to identify and record the periodic and external events that cause processes to do work and record them on the initial process architecture diagram and in process behavior specifications.

4.2.3 Requirements Traceability

The specification of requirements traceability will differ when a CoRE specification is used in place of an RTSA specification. When mapping to an initial process architecture from an RTSA specification, processes trace back to data transformations and control transformations. When mapping to an initial process architecture from a CoRE specification, processes trace back to CoRE artifacts according to Table 9. Note that when Table 9 identifies the relevant artifact as IN', OUT' or REQ', requirements traceability is to the corresponding IN, OUT, or REQ relation of CoRE.

4.3 PROCESS CLUSTERING

The ADARTS process structuring criteria guide the software designer in clustering processes from the initial process architecture to reduce the number of processes. This section describes how the use of a CoRE software requirements specification affects application of the ADARTS process clustering criteria and should be used in conjunction with *Section 8.11 of the* ADARTS Guidebook. There are three kinds of ADARTS process structuring criteria:

- Temporal cohesion (Section 4.3.1)
- Sequential cohesion (Section 4.3.2)
- Functional cohesion (Section 4.3.3)

When you cluster processes, you need to combine the process behavior specifications for the clustered processes. The logic of process behavior specifications identifies the stimuli that activate processes and the action(s) they take in response. Each subsection describes how to modify the process logic for clustered processes according to the clustering criteria applied.

4.3.1 Temporal Cohesion

Temporal cohesion exists for a set of processes when the processes are activated at the same time. You may decide to cluster processes exhibiting temporal cohesion. There are two kinds of temporal cohesion to consider: asynchronous and periodic. Asynchronous temporal cohesion (see Section 4.3.1.1) exists for a set of processes when the processes are activated by the occurrence of the same periodic stimulus. Periodic temporal cohesion (see Section 4.3.1.2) may exist between processes activated by timer (periodic) events.

The criteria described in this section are not unique to clustering processes derived from a CoRE specification; the criteria can be applied to an initial process architecture derived from an RTSA

specification. However, CoRE's precise notation for specifying events and its inclusion timing information and frequency profiles related to events allow you to detect and measure temporal cohesion more accurately.

4.3.1.1 Asynchronous Temporal Cohesion

Asynchronous temporal cohesion exists for a set of processes when the processes are activated by the same periodic stimulus. The stimulus may be:

- An external event (device interrupt)
- The receipt of a message at the same time from another process

The existence of asynchronous temporal cohesion is easily identifiable: two processes are temporally cohesive if each process responds to the same message from a third process or the same event from the external environment. If there is no such common input message or event, asynchronous temporal cohesion does not exist between the processes.

To combine the process behavior specifications of two processes exhibiting asynchronous temporal cohesion, interleave[3] the actions of the response associated with the common stimulus of each process. You should determine and specify whether or not the order in which the actions are performed is significant when combining stimulus-response pairs.

Figures 24, 25, and 26 illustrate examples of the application of asynchronous temporal cohesion from the HAS Buoy case study. Figures 24 and 25 show the process behaviors for Monitor_Location_Correction_Data and Monitor_Incoming_Radio_Messages, respectively. Note that both processes are activated by the detection of the external event Receiver_Interrupt. Figure 26 shows the process behavior for the process that resulted after clustering.

Stimulus	Response
received Receiver_Interrupt	Read (RegisterF) if (RegisterF.Byte_1 = 16#07#) then Location_Correction_Data.u <-- RegisterF.Byte_2 Location_Correction_Data.l <-- RegisterF.Byte_3 send Location_Correction_Data to Determine_Omega_Error

Figure 24. Monitor_Location_Correction_Data Process Behavior

Stimulus	Response
received Receiver_Interrupt	Read (RegisterF) case RegisterF.Byte_1 is when 16#01# => Incoming_Radio_Message.Byte_1 <-- "Red_Light_On" send Incoming_Radio_Message to Determine_Light_Command when 16#02# => -- some logic has been omitted for brevity

Figure 25. Monitor_Incoming_Radio_Messages Process Behavior

3. "Interleave" means to "combine in arbitrary order," not necessarily to "intersperse." The actions of the second process may follow all of the actions of the first process.

Stimulus	Response
received Receiver_Interrupt	Read (RegisterF) case RegisterF.Byte_1 is when 16#01# => Light_Switch <-- 2#1xxxxxxx# write Light_Switch to RegisterH when 16#02# => -- some logic has been omitted for brevity : : when 16#07# => Omega_Error <-- (Lat_Offset <= RegisterF.Byte_2, Lon_Offset <= RegisterF.Byte_3) send Omega_Error to Omega_Queue

Figure 26. Process_Receiver_Interrupt Process Behavior

4.3.1.2 Periodic Temporal Cohesion

Periodic temporal cohesion may exist between processes activated by timer (periodic) events. Unlike asynchronous temporal cohesion, there are varying degrees of periodic temporal cohesion. The greatest degree of periodic temporal cohesion exists between two processes when the periods of the processes are equal and in phase (e.g., each process has a period of 10 seconds, beginning at time t_0). This section describes how to measure periodic temporal cohesion. The guidelines in this section are an optional enhancement to the guidelines in the ADARTS Guidebook. You do not have to follow the guidelines in this section to produce an ADARTS design from CoRE requirements; however, if you do, you will have a more complete understanding of your process structure.

The most efficient use of a single timer event to activate a set of periodic processes that are in phase can be calculated by determining the greatest common divisor (GCD) of the periods (the period of process P is given by t(P)) of the processes ($GCD(t(P_1), t(P_2))$), where P_1 and P_2 are the periodic processes under consideration. If you cluster a pair of periodic processes, you can obtain the same functional behavior by triggering the clustered process at a rate equal to the GCD of the periods of the clustered processes. The greatest degree of temporal cohesion exists between periodic processes P_1 and P_2 when $GCD(t(P_1), t(P_2)) = t(P_1) = t(P_2)$ (i.e., when the processes have equal periods).

However, it may be beneficial to cluster processes with different periods. In this case, $GCD(t(P_1), t(P_2)) = t(P_1) = t(P_2)$ does not hold, implying that there may be situations in which the clustered process will have nothing to do when it is triggered. When you cluster processes with different periods, you should try to maximize the number of times the clustered process does work in response to a timer event (as opposed to responding to the timer event by doing nothing). Let:

Pr(P1) be the probability that process P1 of the cluster will do work in response to a timer event

Pr(P2) be the probability that process P2 of the cluster will do work in response to a timer event

Pr(P1 and P2) be the probability that both P1 and P2 will do work in response to the same timer event

Pr(P1 or P2) be the probability that either P1 or P2 will do work in response to the same timer event

When you select periodic processes (P1 and P2) to cluster, you want to maximize the frequency with which the clustered process will do work in response to the timer event. That is, you want to maximize:

$$Pr(P1 \text{ or } P2) = Pr(P1) + Pr(P2) - Pr(P1 \text{ and } P2)$$

The ADARTS Guidebook states that sequentially cohesive processes cannot be clustered using temporal cohesion. If there is no sequential relationship between P1 and P2, then the probability that one will do work in response to a timer event is independent of the probability that the other will do work in response to the same timer event. Because the individual probabilities are independent, $Pr(P1 \text{ and } P2) = Pr(P1)\,Pr(P2)$ holds. From the discussion above,

$$Pr(P1) = \frac{GCD(t(P1), t(P2))}{t(P1)} \quad \text{and} \quad Pr(P2) = \frac{GCD(t(P1), t(P2))}{t(P2)}$$

implying that the probability of the clustered process doing work in response to a timer event with period gcd(t(P1), t(P2)) is

$$Pr(P1 \text{ or } P2) = \frac{GCD(t(P1), t(P2))}{t(P1)} + \frac{GCD(t(P1), t(P2))}{t(P2)} - \frac{GCD^2(t(P1), t(P2))}{t(P1)t(P2)}$$

For example, consider periodic processes P_1 and P_2 with $t(P_1) = 60$ ms and $t(P_2) = 40$ ms, implying that $GCD(t(P_1), t(P_2)) = GCD(60, 40) = 20$. Therefore, $Pr(P_1 \text{ and } P_2) = 20/60 + 20/40 - 400/2400 = 1/3 + 1/2 - 1/6 = 67\%$, meaning that if P_1 and P_2 are clustered into a periodic process triggered every 20 ms, 67% of the periodic events would cause the process to do work.

For another example, consider processes P_1 and P_2 with $t(P_1) = 10$ ms and $t(P_2) = 20$ ms, implying that $GCD(t(P_1), t(P_2)) = GCD(10, 20) = 10$. Therefore, $Pr(P_1 \text{ and } P_2) = 10/10 + 10/20 - 100/200 = 1/1 + 1/2 - 1/2 = 100\%$, meaning that if P_1 and P_2 are clustered into a periodic process triggered every 10 ms, every periodic event would cause the process to do work. This is an example of the greatest degree of periodic temporal cohesion possible.

The discussion and examples above assume that processes P1 and P2 are in phase. It is possible for two processes to have the same periods but different phases. For example, the timer event for process P1 could be

@T(mon_Time mod 10 ms = 5 ms)

and the timer event for process P2 could be

@T(mon_Time mod 10 ms = 0 ms and mon_Time ≠ 0 ms)

In this case, the timer events of interest to each of the two processes are as follows:

P1: 5 ms, 15 ms, 25 ms, . . .
P2: 10 ms, 20 ms, 30 ms, . . .

This implies that the clustered process should have a period of 5 ms rather than 10 ms and that Pr(P1 and P2) will be 50% rather than 100%.

To combine the process behavior specifications of two processes with the same period, interleave the actions of the response associated with the periodic event in each process. You combine the process

behavior specifications in the same way you do for asynchronous temporal cohesion, except that the stimulus is a periodic event (e.g., @T[mon_Time mod 60 seconds = 0]) rather than asynchronous.

If you cluster processes based on periodic temporal cohesion where the periods of the processes are not equal, it may be necessary to make the responses of the processes conditional. You should determine and specify whether or not the order in which the actions are performed is significant when combining stimulus-response pairs.

Figures 27, 28, and 29 illustrate examples of the application of periodic temporal cohesion for two processes with different periods. Figures 27 and 28 show the process behaviors for processes activated at 10-second and 20-second intervals, respectively. Figure 29 shows the process behavior for the process that resulted after clustering.

Stimulus	Response
when Time_20	Heat_Sensor <– Read (Port_2) send Heat_Sensor

Figure 27. Process Behavior for a 20-Second Periodic Process

Stimulus	Response
when Time_10	Water_Pressure_Sensor <– Read (Port_1) send Water_Pressure_Sensor

Figure 28. Process Behavior for a 10-Second Periodic Process

Stimulus	Response
when Time_10	Water_Pressure_Sensor <– Read (Port_1) send Water_Pressure_Sensor every alternate interval Heat_Sensor <– Read (Port_2) send Heat_Sensor

Figure 29. Process Behavior for the Clustered Periodic Process

4.3.2 Sequential Cohesion

Sequential cohesion exists between two processes when the stimulus activating one process results from an action performed by the other process. You may decide to cluster processes exhibiting sequential cohesion. In addition to reducing the number of processes (and the inherent overhead), clustering based on sequential cohesion reduces the number of messages communicated between processes. An example of sequential cohesion is when one process is activated upon receipt of a message from the other (e.g., an OUT_s process responds to the receipt of the value of an output variable from an OUT_t process).

Assume that two processes, P_1 and P_2, exhibit sequential cohesion because the stimulus that activates P_2 is a receipt of a message from P_1. To combine the process behavior specifications of P_1 and P_2, interleave the actions of P_2 with those of P_1, beginning with the action that causes the stimulus activating P_2. If clustering makes the activating action unnecessary, remove it from the list of actions.

Figures 30, 31, and 32 illustrate an example of the application of sequential cohesion from the HAS Buoy case study. Figures 30 and 31 show the process behaviors for Set_Light_Switch_Value and Control_Light_Switch, respectively. Note that the message Light_Switch is sent from Set_Light_Switch_Value to Control_Light_Switch, causing it to respond. Figure 32 shows a portion of the process behavior for the process that resulted after clustering (additional clustering based on sequential cohesion is reflected in this process logic).

Stimulus	Response
received Red_Light	if (RedLight = "On") then Light_Switch <-- 2#1xxxxxxx# elsif (RedLight = "Off") then Light_Switch <-- 2#0xxxxxxx# send Light_Switch to Control_Light_Switch

Figure 30. Set_Light_Switch_Value Process Behavior

Stimulus	Response
received Light_Switch	write Light_Switch to RegisterH

Figure 31. Control_Light_Switch Process Behavior

Stimulus	Response
received Receiver_Interrupt	Read (RegisterF) case RegisterF.Byte_1 is when 16#01# => Light_Switch <-- 2#1xxxxxxx# write Light_Switch to RegisterH when 16#02# => Light_Switch <-- 2#0xxxxxxx# write Light_Switch to RegisterH when 16#03 => -- some logic has been omitted for brevity . . .

Figure 32. Process_Receiver_Interrupt Process Behavior

4.3.3 Functional Cohesion

When the application of the process clustering criteria based on temporal and sequential cohesion does not reduce the initial process set to a small enough size, you may cluster processes exhibiting functional cohesion. Use the guidelines for clustering processes based on functional cohesion as specified in the ADARTS Guidebook.

To combine the process behavior specifications of two processes exhibiting functional cohesion, either:

- Use multiple stimulus/response pairs to specify process logic
- Use a single stimulus/response pair with a conditional response

Figures 33, 34, and 35 illustrate an example of the application of sequential cohesion from the HAS Buoy case study. Figures 33 and 34 show the process behaviors for Generate_History_Report and Generate_Ship_Detailed_Report, respectively. Figure 35 shows a portion of the process behavior for the process that resulted after clustering.

Stimulus	Response
received (Vessel_Request = "History_Report_Request")	Report.Report_Type <-- "Weather_History_Report" Report.ASCII_Report <-- read Weather_History_Report from the Report_History data store and convert to ASCII send Report to Set_Outgoing_Radio_Message_Value

Figure 33. Generate_History_Report Process Behavior

Stimulus	Response
received (Vessel_Request = "Ship_Detailed_Report_Request")	Report.Report_Type <-- "Ship_Detailed_Report" Buoy_Location <-- get Buoy_Location read Water_Temperature values from Water_Temperature data store calculate term_Averaged_Water_Temperature read Air_Temperature values from Air_Temperature data store -- some logic has been omitted for brevity . . . send Report to Set_Outgoing_Radio_Message_Value

Figure 34. Generate_Ship_Detailed_Report Process Behavior

Stimulus	Response

received Vessel_Request	get next Vessel_Request from Request_Queue
	if (Vessel_Request = "History_Report_Request") then
	Report.Report_Type <-- "Weather_History_Report"
	Report.ASCII_Report <-- read Weather_History_Report from the Report_History data store and convert to ASCII
	else
	Buoy_Location <-- get Buoy_Location
	read Water_Temperature values from Water_Temperature data store
	calculate term_Averaged_Water_Temperature
	read Air_Temperature values from Air_Temperature data store
	-- some logic has been omitted for brevity
	.
	.
	.
	Report.Report_Type <-- "Ship_Detailed_Report"
	send Report to Report_Queue

Figure 35. Generate_Detailed_Reports Process Behavior

4.4 PROCESS COMMUNICATION AND SYNCHRONIZATION

When you mapped from a CoRE specification to an initial process architecture, only data dependencies were recorded (e.g., IN_t processes depend on IN_s processes to supply the values of input

variables). After process clustering is completed, you need to identify how processes communicate and synchronize. For example, you need to identify the processes that initiate message communications and how message communication is managed. Use this section with Sections 8.12 and 3.4.1 of the ADARTS Guidebook.

For each data dependency between processes, choose one of the four kinds of message communications specified by ADARTS:

- Tightly coupled with reply
- Tightly coupled without reply
- First-in, first-out (FIFO) queue
- Priority queue

Also, be sure to record the periodic and external events that cause processes to respond and the data flows representing data communicating with devices on the process architecture diagram and in process behavior specifications.

If a process consumes multiple kinds of messages, consider using a single message communication mechanism, such as a FIFO queue, for communicating all of the messages.

Determine whether all message communications are really necessary. For example, if a periodic IN_s process passes an input variable to an IN_t process, consider only passing the message when:

- The value of the input variable changes
- The IN_t process or a REQ process determines that an updated variable is needed

Subsequent ADARTS activities take into account process logic to optimize the message communication that can reduce the necessary frequency of periodic events. For example, it may be found that an IN_s process that polls an input variable frequently to estimate the latest value of a monitored value may actually need only to respond to an infrequent request for that monitored variable from another part of the system.

4.5 EVALUATION CRITERIA

This section describes how you can evaluate an ADARTS process architecture built from a CoRE software requirements specification. Section 4.5.1 describes how to evaluate process behavior specifications recorded using stimulus/response pairs. Section 4.5.2 describes how to evaluate the timing characteristics of the design. Section 4.5.3 describes how to evaluate the correctness of the design.

As with the enhanced guidelines described in Section 4.2 for specifying process behavior, these guidelines are optional enhancements to the ADARTS method. You do not have to follow the guidelines in this section; however, you should strongly consider following these guidelines to have more confidence in your process architecture.

4.5.1 Evaluating Process Behavior Specifications

You can perform two common analyses using the notation described in Section 4.2.1.2: detection of potential blocking and nondeterminism. To detect potential blocking, collect all stimulus/response pairs for a single stimulus:

Stimulus: E **when** C_1

. . .

Stimulus: E **when** C_N

Ensure that the expression C_1 **or** . . . **or** C_N is always **true** (i.e., true for any assignment of values to variables). The stimulus E will not be recognized when the expression **not** (C_1 **or** . . . **or** C_N) holds. If E refers to an external event, it will be lost; if E refers to an input message obtained via tightly coupled communication, the sending process will be blocked, possibly forever. This analysis is similar to the Completeness Criterion for classes described in Sections 5.3.1 and 5.3.2.

To detect nondeterminism, ensure that the conjunction of any two conditions for the same stimulus is always false (i.e., any combination of values for the variables named in the pair of conditions C_i, C_j results in C_i **and** C_j = **false**). If this is not the case, e.g., if you find two stimulus/response pairs

Stimulus: E **when** C_i
Response: A_i

Stimulus: E **when** C_j
Response: A_j

and there is a situation where C_i **and** C_j holds, then the implementor (or possibly the run-time system) must determine which action will be taken. Such nondeterminism is not always bad, but you should evaluate it to determine if it is really desirable. This rule is similar to the determinism criterion for classes described in Section 5.3.3.

4.5.2 Evaluating Timing Characteristics

Timing analysis can become very complex. Worst case performance is usually assumed to simplify the analysis at the expense of fixing performance problems that may not exist. This section illustrates the kinds of analysis that are possible when the formalisms of CoRE have been used in an ADARTS design.

In the worst case scenario, the frequency of all events is assumed to be at the maximum and the delay of every device and process is at its maximum. When the worst case delay of the input devices, software, and output devices is less than the tolerable delay, the design satisfies the timing requirements:

Input device delay + software delay + output device delay < tolerable delay

However, this analysis assumes that the delay introduced by the devices and the software is linear, i.e., does not depend on the values of variables. This is the simplifying assumption made in CoRE case studies and used in the case study for this report. A more complex analysis would express the delay of each device and process as a function of the inputs and state of the system.

In practice, each stimulus response thread is evaluated to make sure it meets the timing constraint. Each unique event found in the REQ value functions defines the stimulus in a stimulus response thread. For example, in Section App.2.11.1 of the HAS Buoy case study, when event_Periodic_60_Second occurs while INMODE(mode_SOS), the software must broadcast an SOS report. Given a set of priorities, system load, etc., the delays that should be considered include:

- Processing the timer event. There is delay when the operating system wakes up the Generate_Periodic_Reports process (see Section App.3.4.4) or when processing an interrupt from an

external timer. There is also delay if the process is currently handling another stimulus (Mode_Change). Calculate the worst case.

- Set the report type, read Buoy_Location (consider the worst case, could be blocked by another process), format report, and send report to Report_Queue.
- Interprocess communication between Generate_Periodic_Report and Transmit_Reports, including the worst case length of the Report_Queue and the time to prioritize the queue.
- Worst case time for Transmit_Reports to complete sending any page it is in the middle of transmitting.
- Time for Transmit_Reports to create a one-page Outgoing_Radio_Message and write it to the output register.
- The delay introduced by the transmitter device.

Because all the processes in this stimulus response thread run at the highest priority and the message is prioritized to be the highest, the effects of system load are reduced. Of course, other stimulus response threads should be evaluated to see if this higher priority thread causes other deadlines to be missed.

4.5.3 Correctness

This section does not represent a formal system for proving correctness (which is beyond the scope of the report). The example is a walkthrough suggesting a more rigorous definition of correctness based on the precision of process behavior specifications derived from CoRE requirements. Delay is ignored in this example to keep it manageable. Section 4.5.2 suggests the kind of timing analysis that would likely go along with this analysis of correctness.

First, identify a behavior to verify for correctness. For example, in the HAS Buoy case study, Section App.2.11.1, according to the REQ_relation_for_con_Report, the event_Periodic_60_Second (Section App.2.6) when INMODE(mode_SOS) generates an ASCII report of type SOS_Report:

Assume:

`[mon_Time MOD 60 seconds] = 0` **and**
INMODE(mode_SOS)

Prove:

con_Report.Report_Type = "SOS_Report" **and**
con_Report.ASCII_Report = ASCII(term_SOS_Report)

An obvious simplification is helpful: term_SOS_Report consists of only one element, mon_Buoy_Location. Instead, prove:

con_Report.Report_Type = "SOS_Report" **and**
con_Report.ASCII_Report = ASCII(mon_Buoy_Location)

Look for relevant functions and operations that can be used to verify this relation. A good starting place is any process derived from the REQ_Relation_for_con_Report. Section App.3.4.4 describes the Generate_Periodic_Reports process.

Look at INMODE(mode_SOS) and attempt to derive the value of System_Mode. A quick check of the mode machine for HAS_Buoy in Section App.2.7.1 shows that InMode("mode_SOS") can only occur after event_Emergency_Button_Pressed and before event_Reset_SOS. The input device for mon_Emergency_Button, in Section App.2.13.1, is an active device generating an interrupt and a value of 2#1xxxxxxx# for in_Button_Indicator. The process Monitor_Emergency_Button, in Section App.3.4.6, responds to the stimulus by sending a message to Generate_Periodic_Reports, Emergency_Button = "Pressed." A systemwide check reveals that the only other way a message to change the mode can be generated is when event_Reset_SOS occurs. But as indicated, InMode("mode_SOS") precludes this between the event_Emergency_Button and the current time.

According to the logic of the Generate_Periodic_Reports process and the conclusion about messages received, the last response to set System_Mode was:

```
if (System_Mode = "mode_Normal") then
    System_Mode <-- "mode_SOS"
```

Therefore, because there are only two possible values for System_Mode, System_Mode = "mode_SOS."

Looking at the definition for Time_60 and the assumption `[mon_Time MOD 60 seconds] = 0`, you can conclude that the stimulus, when Time_60, occurs shortly after the event_Periodic_60_Second. According to the process logic of Generate_Periodic_Reports, the response to this event should be:

```
Report.Report_Type <--"SOS_Report"
SOS_Report <--read Buoy_Location data store
Report.ASCII_Report <--ASCII(SOS_Report)
send Report to Report_Queue
```

An evaluation reveals that the Buoy_Location has been updated sometime within the last 30 seconds. Because the buoy does not change location quickly (Section App.2.10.2), ignore the age of the location data and conclude that the Report_Queue now contains a report with the following values:

```
Report = ("SOS_Report," ASCII(Buoy_Location))
```

Following the trail (i.e., stimulus/response thread) shows that the process Transmit_Reports in Section App.3.4.7 takes reports from the Report_Queue. Because the queue is prioritized and SOS_Reports get highest priority, assume (as opposed to doing throughput analysis) that no other SOS_Reports are on the queue and this is the next one processed.

With a little detailed evaluation of the process logic, the Page_Count = 1 and

```
write Outgoing_Radio_Message to RegisterG, where
    Report_Code <-- 2#10000001#
    Page_Count <--2#00010001#
    Bytes_3-512 <-- ASCII(Buoy_Location)
```

Finally, the transmitter described in Section App.2.11.2 sends the report:

```
con_Report.Report_Type = "SOS_Report" and
con_Report.ASCII_Report = ASCII(mon_Buoy_Location)
```

4.6 FUTURE WORK

There is still a great deal of potential in several areas to exploit the precise specification of behavior:

- The attempts at verification, although more rigorous, are still not formal. The initial work in Section 4.5 shows promise.
- There may be more analytical approaches to deriving ADARTS stimuli from CoRE events. These approaches could begin with the initial mapping and proceed through the design activity with correctness preserving transformations (similar to the way process clustering is applied). Moving stimuli from one process to another with appropriate updates to process logic and message communication is possible.
- Further work will allow development of guidelines for allocating timing requirements derived from timing requirements associated with REQ relations to a series of IN_s, IN_t, REQ, OUT_t, and OUT_s processes that make up a stimulus response thread.

5. CLASS STRUCTURING

In the class structuring activity, you develop the static view of the software design. To begin class structuring, you should have a complete CoRE specification for software requirements, descriptions of customer-mandated external systems, and a knowledge of the implementation environment. You will use this information to perform the activities in class structuring. As you apply the guidelines in this section, you should think carefully before mapping requirements in different CoRE classes to the same ADARTS class because the requirements are likely to change independently of each other.

5.1 DERIVING CLASSES

Table 1 is an overview of how you form the abstractions that form the basis for ADARTS classes. The abstractions are described in detail in Section 11.4 of the ADARTS Guidebook. Classes are derived from variables (e.g., monitored and controlled) and from relations (e.g., REQ and NAT). In general, you will use variables and terms to derive data abstraction and collection classes and IN and OUT relations to derive device interface and external system classes. The following subsections describe how you derive ADARTS classes from these parts of the CoRE behavioral model.

5.1.1 Device Interface Classes

Create one device interface class for each unique kind of device with which the software will interface. Devices are mentioned in the definitions of input and output variables, which appear in CoRE boundary classes. Map each input variable and output variable to a device interface class. The mapping will not be one-to-one for cases in which a single device is associated with multiple input or output variables. You should use the guidelines described in this section with Section 9.4.1 of the ADARTS Guidebook.

You should create one object for each device that will be controlled by the software. There will be at least one object for each device interface class. There will be multiple objects for the same class if there are several devices of the same type.

Table 10 summarizes the services encapsulated by device interface classes. In general, you will create one operation for each activity. If a device interface class contains operations to approximate monitored variables or output variables, it should have separate operations to read input variables from the device or to write output variables to the device. These operations should remain separate because of the possibility that they will be invoked by different processes.

Table 10. Activities Encapsulated by Device Interface Classes

Service	Encapsulated in Device Interface Class
Read input variables	Always
Write output variables	Always
Operate the device	Always
Approximate monitored variables from input variables	If all inputs for a monitored variable come from the same device and there is no need to store a collection
Generate output variables from approximations of controlled variables	All outputs for a controlled variable go to the same device and there is no need to store a collection

Do not include translation of input variables to monitored variable approximations or output variables from controlled variable approximations unless the resulting class will be cohesive and understandable. Use a computation class for the translation if you do not include it in the device interface class.

You should consider possible changes before you use a single class to interface with a device and approximate monitored variables. For example, in the HAS Buoy case study, water temperature is calculated by the Water Temperature Comp Class instead of the Water Temperature Device Interface Class because the number of water temperature sensors may change in the future. On the other hand, the number of air temperature sensors is not expected to change, so approximation of the Air Temperature Sensor monitored variable is performed by the Air Temperature Sensor Device Interface class (see Figure 36).

You should also be certain that a device interface class does not encapsulate multiple concerns. For example, if the algorithm for computing air temperature was sufficiently complex, you could reasonably consider it a concern separate from operating the air temperature sensor device. In this case, you would encapsulate the algorithm in a separate computation class, even if the number of devices and the algorithm were not expected to change independently.

Examples of device interface classes appear in Figure 36.

5.1.2 External System Classes

Create an external system class to encapsulate details of interfaces between your system and other hardware/software systems mandated by the requirements as described in Sections 9.4.2 and 5.1.1.2 of the ADARTS Guidebook. The definitions of input and output variables, which appear in CoRE boundary classes, will indicate if they are produced or consumed by external systems. Textual annotations will indicate if any of the expressions in REQ, IN, and OUT tables are to be computed by external systems.

5.1.3 Data Abstraction Classes

Data abstraction classes encapsulate concerns related to the representation of data. Use this section with Section 9.4.3 of the ADARTS Guidebook. You should map each monitored variable, controlled variable, input variable, and output variable to a data abstraction class. The mapping usually is not one-to-one; variables of the same type will be mapped to the same class unless you expect the types

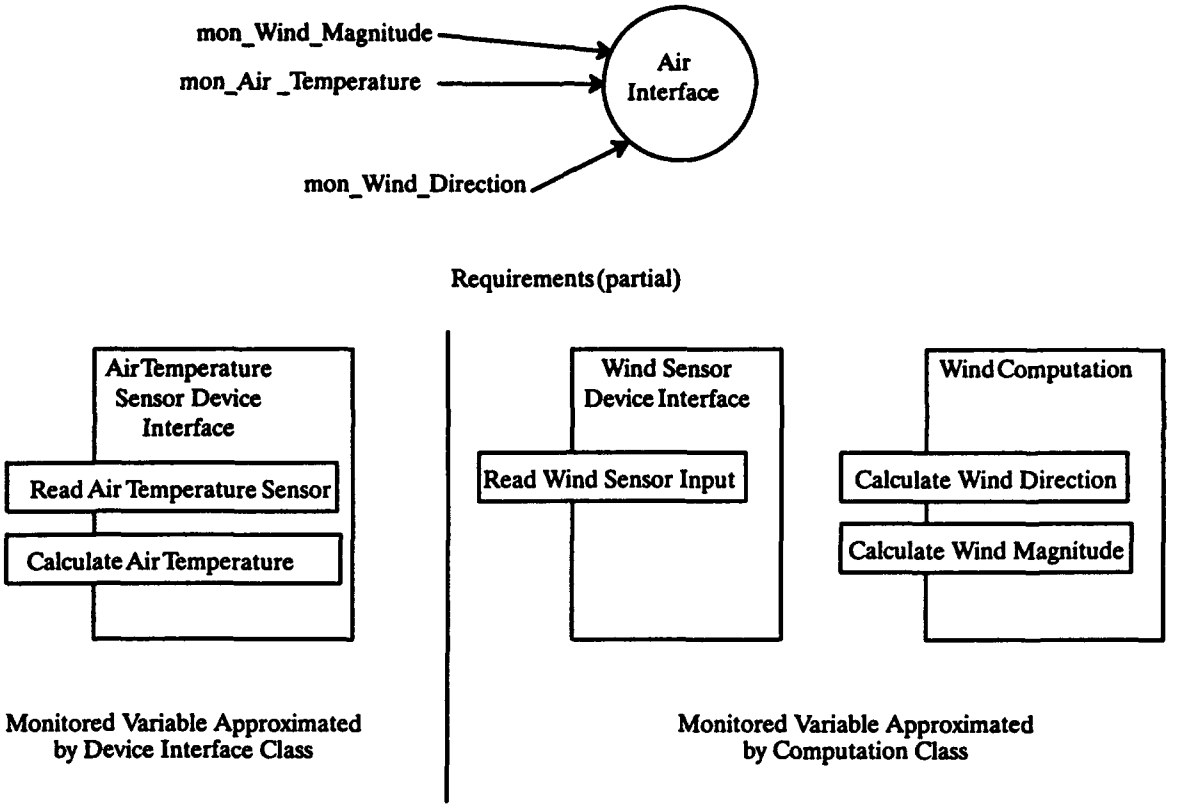


Figure 36. Examples of Device Interface Classes

to change independently. If a variable is of a collection type, you will map it to a collection class (to encapsulate concerns about the collection as a whole) and to a data abstraction class (to encapsulate concerns about one member of the collection).

You should consider mapping each term to a data abstraction class. However, it is not necessary to create a data abstraction class for every term. Terms are included in the CoRE method for the convenience of the requirements analyst. A term is simply a named expression. A very simple term (such as the inverse of a monitored variable) could be mapped to an operation on an existing class (such as the data abstraction class for the monitored variable). A term representing a complex computation (such as a trigonometric function) could be mapped to a computation class.

Recall that environmental variables are defined in CoRE boundary and term classes. Terms can be defined in CoRE boundary, term, and mode classes. These are the CoRE classes that contain the requirements you use to define ADARTS data abstraction classes.

A data abstraction class is associated with a single copy of some type of information; you use data collection classes (see Section 5.1.4) to represent collections of two or more items of information. For example, the HAS Buoy case study requirements define term_Averaged_Water_Temperature as the arithmetic average of the past six water temperature readings. The collection of readings would be mapped to a data collection class; you would form a data abstraction class for individual water temperature values.

Create one or more data abstraction classes for each unique value type associated with a variable or term in the CoRE specification. If two CoRE variables are of the same type but you expect their types

to change independently, create separate data abstraction classes. If the value type is composed of a collection of identical simpler types (e.g., a set of sensor readings), create a data abstraction class for the simple (i.e., single-valued) types and a data collection class for the collection. If two variables have the same type but you expect the types to change independently, create a data abstraction class for each. If the variables are of a collection type, create two data abstraction classes if you expect the underlying single-valued types to change independently.

Note that a single-valued type may be decomposable into several atomic values. For example, the input variable in_Wind_Sensors has four atomic values (i.e., the sensor readings for the north, south, east, and west wind sensors). In general, you should map such a type to a single data abstraction class providing operations to set and retrieve individual atomic values. If you expect some of the atomic values to change independently of others, you may choose to map the attributes to separate classes.

When developing data abstraction classes, you should remember that the types and units associated with variables and terms in a CoRE specification are not requirements and you are free to use different types and units in the design. For example, you could represent ~mon_Water_Temperature in degrees Fahrenheit even though mon_Water_Temperature is expressed in degrees Celsius.

Create one object for each approximation (e.g., ~mon_Wind_Speed) that the software will maintain. For approximations that are collections, you will create an object for the entire collection and one or more objects for single elements of the collection.

To determine the operations on a data abstraction class, examine the expressions in which the corresponding variables appear. Expressions appear in REQ, IN, and OUT relation tables and in term definitions. You may choose to create operations specifically for simple expressions, such as increment and decrement. If an expression is complex or requires an algorithm that is subject to change (such as an iterative approximation algorithm), consider mapping the expression to a separate computation class (see Section 5.1.7). You may choose to create a computation for part of an expression (such as a trigonometric function) and establish a dependency between the data abstraction class and the computation class. If two or more variables are mapped to the same data abstraction class and the expressions are significantly different, you may want to consider mapping the variables to separate data abstraction classes.

Figure 37 contains an example of two sets of similar requirements and the corresponding data abstraction classes. Both air temperature and water temperature are provided by the environment and averaged over a period of time. In each case, the monitored variable containing a single temperature reading and the term representing the average are mapped to the same data abstraction class. However, it is possible that the range of values, precision, or other characteristics of the two temperatures will change independently, necessitating separate classes.

In Figure 38, mon_Buoy_Location is an example of a variable whose value is composed of two atomic values (latitude and longitude). The data abstraction class created for mon_Buoy_Location has separate read and set operations for latitude and longitude.

5.1.4 Data Collection Classes

Certain requirements are stated in terms of collections (e.g., sets, sequences) of values of the same type. For example, in the HAS Buoy case study, the terms for averaged air temperature and water temperature and average wind direction and magnitude (speed) are defined as arithmetic averages of multiple samples

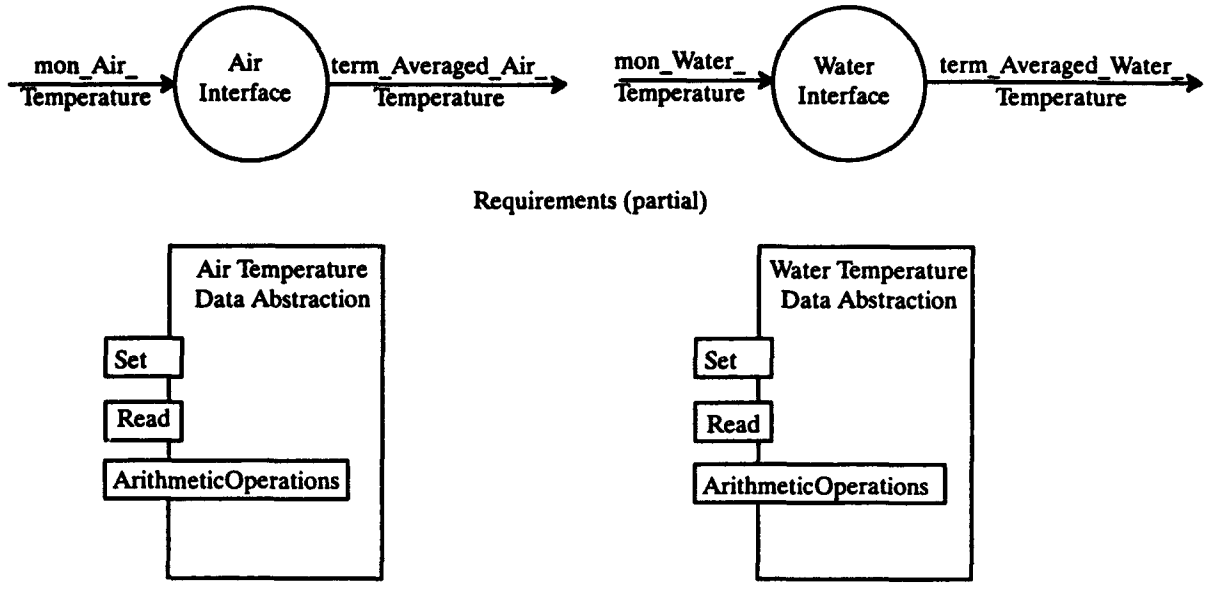


Figure 37. Example of Data Abstraction Classes

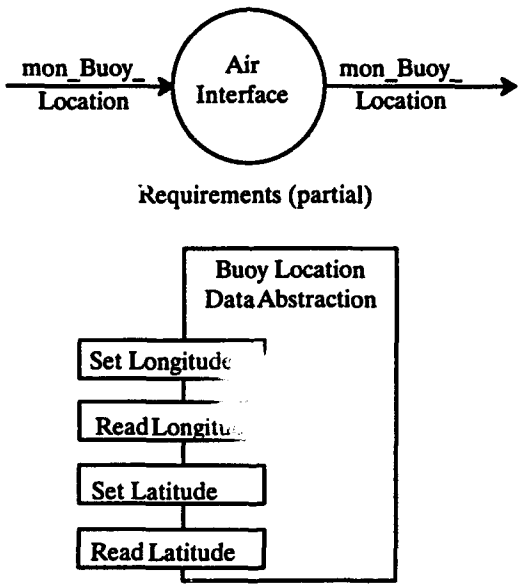


Figure 38. Example of Data Abstraction With Multiple Atomic Values

of the values of the corresponding monitored variables. Search through the REQ, IN, and OUT relations in CoRE boundary classes for expressions that refer to sets, sequences, or other collections. Often, these expressions will refer to collections of monitored variables or terms taken over a period of time. You should also look for these expressions in the definitions of terms, which can appear in CoRE boundary and term classes. Map each such expression to a data collection class, which will export operations on the collection as a whole. Examples of operations are iterating (i.e., examining the collection one item at a time), sorting the collection, and searching for items with specific

properties. A collection class deals with the entire collection; you will create a data abstraction class (see Section 5.1.3) to deal with individual items in the collection. As described in Section 9.4.4 of the ADARTS Guidebook, ADARTS maintains a separation of concerns between data abstraction and collection classes because they can easily change independently of each other.

To find expressions that refer to collections, look for set operators, such as summation ("SUM" in the HAS Buoy case study), or for direct references to sets (e.g., "{i: 0<=i<=5: mon_Temperature(i)}"). For example, term_Averaged_Air_Temperature, illustrated in Figure 37, is defined with the following expression:

```
ROUND[(SUM(i:0<=i<=5:mon_Air_Temperature(t - 10·i))) / 6]
```

The parameter t refers to the current time. The entire expression refers to the set of six values of the monitored variable mon_Air_Temperature, sampled at intervals of 10 seconds, with the most recent sample being the current value of mon_Air_Temperature. The significance of this expression for Class Structuring is that the software must maintain a collection of historic values of mon_Air_Temperature to approximate term_Averaged_Air_Temperature[4]. The collection class corresponding to this expression is shown in Figure 39.

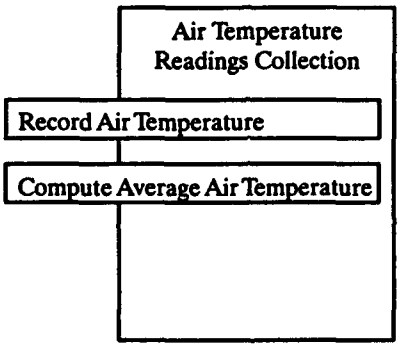


Figure 39. Collection Class for Air Temperatures

Map to a collection class each variable, term, or expression that implies need for a collection of similar items. Collection classes may be required for inputs and outputs as well as monitored and controlled variables, e.g., input data read in bursts and the input variable defined as a sequence of values.

Create an object for each collection, not for each item in each collection.

5.1.5 State Transition Class

An ADARTS state transition class (see Section 9.4.5 of the ADARTS Guidebook) hides the contents of a CoRE mode machine. Mode machines are defined in CoRE mode classes. You should map to a state transition class each unique mode machine in the requirements specification. (Two mode machines are identical if their definitions are the same.) Each mode associated with the mode machine becomes a state of each object derived from state transition class. The possible changes to

4. The expression defines the value of term_Averaged_Air_Temperature, assuming no delay or error. The term is used to define a controlled variable representing a report transmitted in response to an event. Because the software cannot guarantee that the most recent temperature sample is taken when the event occurs, the requirements must limit how old the temperature sample is allowed to be. See the CoRE Guidebook and Section 4 of this report for more information.

a state transition class include the addition of new states and changes to the transitions between states. The ADARTS Guidebook mentions the possibility of change to the sequence of actions taken in response to an event. However, CoRE mode machines do not associate actions with mode transitions.

In general, you should create an operation for each event that causes the mode machine to change states. Section 4.1.3.1 of the CoRE Guidebook states that an event occurs when a condition changes value. Events can be given names such as event_Button_Pressed and are described using event expressions in the form of

```
@T(C1) when C2
```

or

```
@F(C1) when C2
```

where C_1 and C_2 represent conditions and the "when" part of the expression is optional. Look for named events or expressions such as these in the definition of a mode machine, and map them to operations on the corresponding state transition class. If a large number of events are associated with a single mode machine, you may choose to map several events to a single operation. If the operations for events do not return the current mode, the state transition class must include an operation to query the current mode. Trace the query operation to tables that mention modes of the mode machine and expressions that include the subexpression

```
INMODE(X)
```

where X is one of the modes of the mode machine.

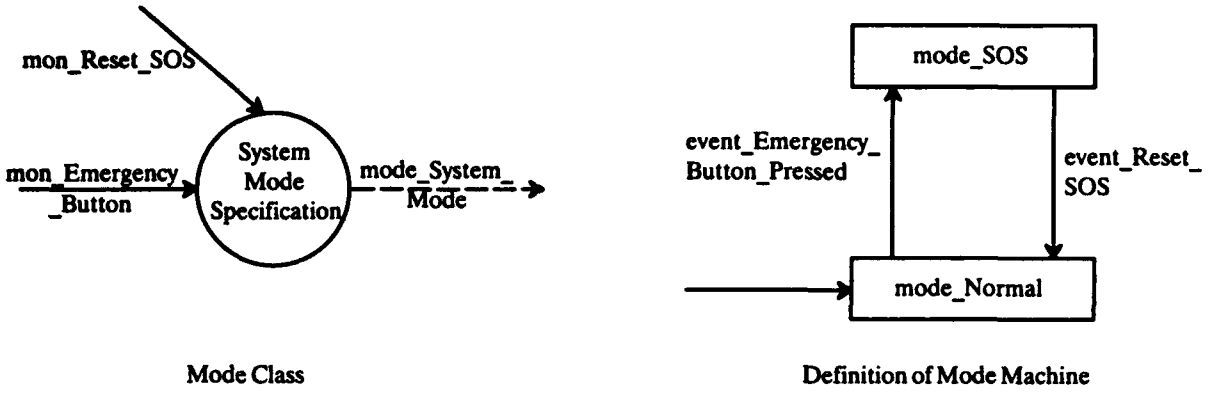
Figure 40 is an example of a mode machine, its definition in terms of modes, transitions, and events, and the corresponding ADARTS state transition class.

Create one object for each mode machine in the CoRE specification. This will almost always amount to creating one object for each state transition class. However, if the requirements specification included two or more identical mode machines and you mapped them to the same state transition class, then you will create multiple objects from a single state transition class.

5.1.6 User Interface Class

The purpose of a user interface class is to hide the look and feel of an interface between your application and a human user. Look and feel requirements are more abstract than and can change separately from input and output requirements. Examine REQ tables and the definitions of terms to find user interface look and feel requirements, and map these requirements to a user interface class. Requirements for ADARTS user interface classes will generally come from CoRE boundary classes. Use this section with Section 9.4.6 of the ADARTS Guidebook.

For example, the Fuel Level Monitoring System case study in the CoRE Guidebook includes requirements for displaying three operator messages. One of the messages is a number representing the level of fuel in a tank. The other two messages are textual warnings that flash on and off at a rate of 1 Hz (see Section B.8 of the CoRE Guidebook). Requirements for the operator display include position and format for the messages and the flash rate for the two warning messages. This information can change independently from the OUT relation, which specifies how the software causes the alarm to sound.



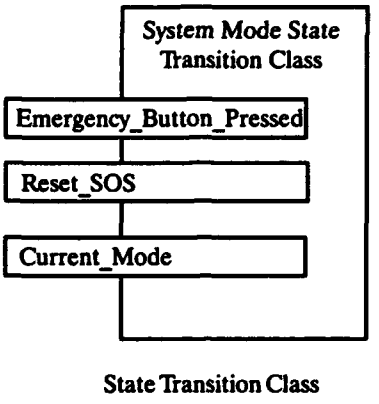


Figure 40. Example of State Transition Class

You could map the user interface requirements to a user interface class as shown in Figure 41. The Set_con_Level_Display operation displays its numeric parameter (an approximation of mon_Fuel_Level in the part of the screen allocated to con_Level_Display. The operations Set_con_High_Alarm and Set_con_Low_Alarm each take a Boolean parameter indicating if the corresponding message should be visible. A process would call these operations often enough to achieve the 1 Hz blink rate. This class would depend on device interface classes that would encapsulate low-level details of the audible alarm and display screen.

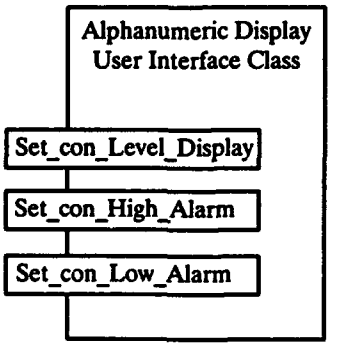


Figure 41. Example of User Interface Classes

5.1.7 COMPUTATION CLASS

Computation classes as described in the ADARTS Guidebook (Section 9.4.7) encapsulate computational algorithms and execution sequences. Computation classes derived from a CoRE specification encapsulate only computational algorithms because CoRE requirements are not stated in terms of imperative actions.

Map to a computation class requirement implying the need for a computational algorithm that is sufficiently complex to justify a separate class or that can change independently from concerns allocated to other classes. It is not necessary to map every computation to its own class. The Air Temperature Sensor device interface class in Figure 36 includes a computation (i.e., the conversion of input variable approximation to a monitored variable approximation) that is simple and not expected to change independently of the other concerns related to the device interface. The Wind Sensor device interface class and Wind Computation classes in the same figure exemplify a computation that should be encapsulated in a separate class because of its complexity and the possibility of its changing separately from the device interface concerns.

Search the REQ, IN, and OUT tables, mode transition tables, and term definitions for complex expressions and subexpressions indicating the need for a computational algorithm. You can form ADARTS computation classes from all kinds of CoRE classes (boundary, term, and mode). You should also examine term definitions. Map each expression to a computation class. You may choose to map an expression to one computation class and a subexpression to another, and you also may choose to map related computations to the same class. You may even have the opportunity to use one function provided by the class in the definition of another. For example, the function encapsulated by Wind Computation Class is

```
Wind_Direction=   ARC_COS(Wind_Velocity_X_Axis/Wind_Magnitude)
```

where

```
Wind_Magnitude=   SQRT(    Wind_Velocity_X_Axis**2
                       +   Wind_Velocity_Y_Axis**2)
```

In the case study, the definitions of Wind_Direction and Wind_Magnitude were mapped to Wind Computation Class, and the ARC_COS function was mapped to a separate Trigonometric Functions Computation Class.

Create at least one object for each computation class. If the functions provided by a computation class are defined solely in terms of their parameters, then the class will not encapsulate any state information and a single object will be sufficient. On the other hand, a function computed incrementally (such as a running total) will require state information. In this case, you may need to define additional objects.

5.2 ABSTRACT INTERFACE

This section describes how you can maintain CoRE's level of precision in ADARTS class structuring work products. Section 5.3 explains how you can use the abstract interface to verify some important characteristics of classes. See the ADARTS Guidebook (Section 9.5) for a complete introduction to the abstract interface. The guidelines in this section are optional enhancements to the ADARTS

method facilitated by the precision of CoRE requirements and motivated by the desire to maintain CoRE's level of precision in the design. The benefits of precision are discussed in Section 2.5. You do not have to follow the guidelines in this section, but you should strongly consider doing so.

The purpose of the abstract interface is to record information about a class that is unlikely to change over the life of the software. The abstract interface is the part of the class specification that can be used by other software in the system. The rest of the class specification is considered "hidden" from the viewpoint of other software. Information hidden by a class can be changed without affecting other classes or processes in the system.

You make use of five basic concepts to document the abstract interface precisely. These concepts are abstract state, operations, invariants, preconditions, and postconditions.

5.2.1 Abstract State

This is an abstraction of information maintained by objects derived from a class. An example of an abstract state is the set of air temperature readings maintained by the Air Temperature Readings Collection object (see Section App.4.8). Objects derived from some classes (e.g., the Trigonometric Functions Computation Class) maintain no information and will have no abstract state. Objects derived from state transition, data abstraction, and collection classes will always have an abstract state. The abstract state of a device interface object may describe some characteristic of the device that is significant to the software.

Describe the abstract state textually and give it a name. You will use the name to define invariants, preconditions, and postconditions. In some cases, the abstract state will have attributes that you will want to distinguish by assigning each a name. For example, the abstract state of the Buoy Location Data Abstraction object has attributes Latitude and Longitude. You should also describe the domain of values that the abstract state can assume. If the abstract state comprises several attributes, you should associate a domain of values with each. Whenever possible, you should take the name and domain of values from the requirements mapped to the class. If you find that the abstract state of two objects derived from a class have different value domains, you should consider deriving the objects from separate classes. Table 11 contains examples of abstract states.

Table 11. Examples of Abstract State

Class	Abstract State	Initial Value
SOS Report Data Abstraction	state_Latitude (value of ~<Latitude>mon_Buoy_Location) state_Latitude (value of ~<Longitude>mon_Buoy_Location) state_Latitude_Defined (Boolean value—TRUE if Set_Latitude operation called at least once) state_Longitude_Defined (Boolean value—TRUE if Set_Longitude operation called at least once)	state_Latitude_Defined=FALSE state_Longitude_Defined=FALSE
Air Temperature Readings Collection	state_Collection (Value: A set of up to 6 elements. The elements are taken from the same domain as mon_Air_Temperature.)	{} (i.e., the empty set)

You should specify the initial value of the abstract state in enough detail to allow you to predict how the operations will behave. It is not always necessary to specify the initial value of each attribute of

the abstract state. For example, the SOS_Report Data Abstraction class encapsulates the format of the 60-second SOS report, which contains the current buoy location. The abstract state of the object derived from this class consists of four items: two numeric values for latitude and longitude, and two Boolean flags that indicate whether the numeric values have been defined. The behavior of the operation returning the ASCII encoding of the SOS report is defined in terms of the Boolean flags, and returns an error if either latitude or longitude is not defined. In this case, you can predict the behavior of each operation without specifying an initial value for latitude and longitude, as long as the Boolean flags are initially false.

5.2.2 Operations

Operations are the services exported by objects to the rest of the software. Some operations alter and report the abstract state of an object; others, such as trigonometric functions may just compute a value based on their parameters. You should name each operation and describe its parameters (if any). Informally describe the effect of each operation; you will also describe operations formally using preconditions and postconditions.

5.2.3 Invariants

Invariants are assertions about the abstract state of the class. Invariants are always true. That is, an invariant is true initially, and no change to the state of the system will ever negate it. Wherever possible, express invariants as logical expressions. You will use invariants to evaluate the class specification. You will also use invariants, along with preconditions and postconditions to evaluate the software architecture. Table 12 contains the invariants for the classes mentioned in Table 11.

Table 12. Examples of Invariants

Class	Invariants
Buoy Location Data Abstraction	This class has no invariants
Air Temperature Readings Collection	SIZE(state_Collection) ≤ 6

It is possible to write global invariants which relate the abstract state of objects derived from one class to the abstract state of objects derived from other classes, or to requirements. This technical report deals only with local invariants, which assert properties of a single class.

5.2.4 Preconditions and Postconditions

Preconditions and postconditions are assertions about the abstract state which define the behavior of operations. As with invariants, the preconditions and postconditions described in this report deal only with the abstract state and parameters. They do not mention other classes or requirements. If a precondition holds when an operation is invoked, the associated postcondition will hold from the time the operation completes until the next change to the abstract state or one of the parameters. Often, an operation will behave differently under different circumstances. In such cases, you will use one precondition-postcondition pair to describe each type of behavior. For example, the Compute Average Air Temperature Operation on the Air Temperature Readings Collection Class returns the

arithmetic average of the air temperature readings if there are six readings in the collection, and an error if there are fewer than six. The error indication is considered an undesired event (see Section 9.5.3 of the ADARTS Guidebook).

As with invariants, you should write preconditions and postconditions as logical expressions. When the operation changes the value of the abstract state or a parameter, use a naming convention to distinguish the original value from the value upon completion of the operation. In the examples, this report uses the prefix "Updated_" to identify the value upon completion. Table 13 contains the preconditions and postconditions for the Record Air Temperature Operation of the Air Temperature Readings Collection Class, where state_Collection represents the abstract state of a collection and Value is a parameter to the operation.

Table 13. Preconditions and Postconditions for Record Air Temperature Operation

Precondition	**Postcondition**
SIZE(state_Collection)<6	Updated_state_Collection =state_Collection UNION {param_Value}
SIZE(state_Collection)=6	Updated_state_Collection=state_Collection – OLDEST(state_Collection) UNION {param_Value}

When defining the behavior of state transition classes, you must specify what happens when an event occurs in a mode not anticipated in the CoRE specification. For example, the mode machine in Figure 40 does not specify what happens if event_Emergency_Button_Pressed occurs in mode_SOS or if event_Reset_SOS occurs in mode_Normal. It is reasonable to assume that nothing should happen in either case. There certainly should not be a mode change. Also, detection of either event/mode pair by the software should not be considered an error because both can happen. In general, if you can prove that an event/mode combination not mentioned in the requirements will never occur, then the state transition class can legitimately report an error when the corresponding operation is invoked in the corresponding state. Otherwise, the operation should do nothing when invoked in that state.

Where feasible, you should refer to the requirements when writing preconditions, postconditions, invariants, and defining the abstract state. This will minimize the configuration control problem that results from changes in requirements. For example, the Calculate_Air_Temperature Operation on the Air_Temperature_Sensor Device Interface Class references IN'_for_mon_Air_Temperature (see Section App.4.1.1).

You should specify error bounds for operations that can introduce error. Error is usually introduced by computations on real numbers. Error is inherent in any software representation of real numbers because the precision of the representation is limited by the number of bits available. On the other hand, representations of discrete values (e.g., Boolean and enumerated values) do not necessarily introduce error. Any operation producing a value (i.e., a return parameter or update to the abstract state) that is not taken from a discrete set has the potential to introduce error, and you should specify a bound on the error as part of the postconditions. An example of an operation that can introduce error is the Compute_Averaged_Air_Temperature Operation on the Air_Temperature_Readings Collection Class, shown in Table 14. In the first postcondition, Averaged_Air_Temperature is the value returned by the operation, and the maximum error allowed is 1 degree centigrade. This means that, upon completion of the operation, the return parameter Averaged_Air_Temperature will differ from ROUND[SUM(Collection)/6] by a maximum of 1 degree. Stated formally:

|Averaged_Air_Temperature − ROUND[SUM(Collection)/6]| ≤ 1 degree centigrade

Note that the error bound relates a value returned by an operation to the abstract state of the class. It does not attempt to relate the value returned or the abstract state to the environmental entity that it represents (in this case, term_Averaged_Air_Temperature). This is consistent with the purpose of the error bound, which is to limit the error introduced by the operation. In software architecture design, you will relate the return value Averaged_Air_Temperature to the approximation of term_Averaged_Air_Temperature. No error bound is specified for the second postcondition because no value is returned and no abstract state is updated.

Table 14. Example of Bounding Error

Precondition	**Postcondition**
SIZE(state_Collection)=6	Averaged_Air_Temperature = ROUND[SUM(state_Collection)/6] Maximum Error: 1 degree centigrade
SIZE(state_Collection)<6	ERROR(Insufficient Data)

5.3 EVALUATION CRITERIA

The criteria for evaluating class structuring work products discussed in Section 9.10 of the ADARTS Guidebook applies to classes and objects derived from CoRE requirements. This section discusses some additional criteria that you can apply if you specified the abstract interfaces as described in Section 5.2 of this report. This version of the evaluation criteria does not take error bounds into consideration.

This section explains how you can use enhancements to the abstract interface described in Section 5.2 to verify some important characteristics of classes. As with the enhanced abstract interface guidelines, these guidelines are optional enhancements to the ADARTS method. You do not have to follow the guidelines in this section; however, you should strongly consider doing so if you followed the guidelines in Section 5.2.

Sections 5.3.1 through 5.3.5 provide some simple rules for ensuring completeness, self-consistency, and correctness of a class specification. These rules deal with classes in isolation; they do not describe how to ensure that a class is consistent with other classes, with the processes that use it, or with requirements. You will use the software architecture design evaluation criteria to evaluate consistency between classes and processes and consistency of the software design with requirements. Section 5.3.6 contains some simple rules for ensuring that you have defined all the classes and operations necessary to satisfy requirements. Section 5.3.7 discusses error analysis.

Consistency between classes is a topic that you should address during implementation. In class structuring, you derive the dependency graph by making assumptions about how you will implement the internals of each class. If you identify a dependency between two classes, you have assumed that the implementation of one class will use the abstract interface of the other and that the abstract interface will be adequate. You cannot verify the assumption because you do not know how classes will use each other. On the other hand, you can verify consistency between processes and the classes they use because you developed process behavior specifications during process structuring.

The examples in this section are very detailed and are developed using formal logic. The purpose is to illustrate the principles involved—not to imply that your evaluation of your classes must be this detailed or this formal. Gries (1981) contains a good discussion of the concepts motivating this section and provides some very good guidance for verifying implementations. The notation used in this section is defined in Section 2.7.

5.3.1 Completeness Criterion—Strong Form

Each precondition describes a scenario in which an operation can be invoked. The corresponding postcondition describes the result of invoking the operation under the scenario. If an operation is invoked in some situation not described by any precondition, then it is not possible to predict what the operation will do.

Every situation in which an operation can be invoked should be described by at least one precondition. The weakest precondition for an operation should describe all permissible values of the abstract state and parameters to the operation. The weakest precondition is formed by logically disjoining (i.e., or-ing together) the individual preconditions[5]. You can be certain of completeness if the weakest precondition describes all possible values of the abstract state and parameters. Stated more formally, where $P_1, P_2, \ldots, P_L$ are preconditions for an operation, an operation satisfies the strong completeness criterion if the following is true for all values of the abstract state and parameters:

P_1 **or** P_2 **or** . . . **or** P_L

P_1 **or** P_2 **or** . . . **or** P_L forms the weakest precondition for the operation. If the operation is invoked when this condition is false, then you may not be able to predict what the operation will do. This form of the completeness criterion is somewhat more restrictive than it needs to be because it covers values of the abstract state that may be disallowed by the invariants. Section 5.3.2 discusses a less restrictive form that you can use in place of this one.

For example, the preconditions to the Record Air Temperature operation on the Air Temperature Readings Collection Class described in Section App.4.8 are:

Precondition 1 for Record Air Temperature operation: SIZE(state_Collection)<6

Precondition 2 for Record Air Temperature operation: SIZE(state_Collection)=6

Thus, where P_1 and P_2 respectively denote Preconditions 1 and 2,

P_1 **or** P_2 ≡ SIZE(state_Collection)<6 **or** SIZE(state_Collection)=6
≡ SIZE(state_Collection)≤6

which does not hold for all values of state_Collection. The situation not covered by the weakest precondition is **not**(SIZE(state_Collection)≤6) ≡ SIZE(state_Collection)>6.

If the operation is called when this condition holds, you cannot predict the outcome. At this point, you can take one of two actions. One possibility is to examine the invariants to determine if the Record

5. As the term is defined in Gries (1981), this is the weakest precondition with respect to the disjunction of the postconditions for the operation. It describes all values of the abstract state and parameters for which at least one of the postconditions will hold when the operation completes.

Air Temperature operation can ever be called when SIZE(Collection)>6. Section 5.3.2 tells you how to do this. The other possibility is to make the operation more robust by changing the preconditions to allow for this condition. You could broaden the second precondition to

SIZE(Collection) ≥ 6

or you could leave the second precondition alone and add a third:

Precondition 3 for Record Air Temperature operation: SIZE(state_Collection)>6

Postcondition 3 could describe the operation returning an error condition or deleting enough old elements of the collection to reduce the collection size to six.

In general, when you apply this form of the Completeness Criterion and the weakest precondition is not **true**, you should examine its logical negation. Either apply the other form of this criterion to convince yourself that the operation will never be called when the weakest precondition does not hold, or change the set of preconditions to include the logical negation of the weakest precondition.

5.3.2 Completeness Criterion—Weak Form

Section 5.3.1 discussed a form of the Completeness Criterion that may sometimes require you to specify the behavior of an operation under a scenario that you do not expect to happen. The criterion in that section has the advantage of being easy to use, but it may require you to change existing preconditions or to add new ones. This section discussed a form of the same criterion that allows you to use invariants to eliminate scenarios that will never arise.

The weak form of the Completeness Criterion can be stated as follows, where P_1, P_2, . . ., P_L are preconditions for an operation, and I_1, I_2, . . ., I_M are invariants associated with the class:

I_1 and I_2 and . . .and I_M implies P_1 or P_2 or . . . or P_L

All of the invariants are true all of the time. Because the invariants discussed in this report are maintained by the class, you can establish this using the Initial State Criterion (see Section 5.3.4) and the Consequence Criterion (see Section 5.3.5). Because the abstract state will never assume a value that does not satisfy I_1 **and** I_2 **and** . . .**and** I_M, this value does not need to be covered by any of the preconditions.

The weak form of the Completeness Criterion says that whenever the invariants are all true, at least one of the preconditions must be true. This criterion does not prohibit the preconditions from covering situations that are not allowed by the invariants; it simply states that the preconditions may not omit any situation that the invariants allow.

Application of the strong form of the Completeness Criterion to the Record Air Temperature operation on the Air Temperature Readings Collection Class necessitated either rewriting one of the preconditions or adding a new one. The weak form of the criterion does not require any such change. The invariant for the class is SIZE(Collection) ≤ 6, and the preconditions to the operation are SIZE(Collection)<6 and SIZE(Collection)=6. The weak form of the Completeness Criterion for this operation is satisfied if:

SIZE(state_Collection) ≤ 6 implies SIZE(state_Collection)<6 or SIZE(state_Collection)=6

This is the same as

SIZE(state_Collection) ≤ 6 **implies** SIZE(state_Collection) ≤ 6

which is obviously true.

5.3.3 Determinism Criterion

If you use multiple preconditions to describe different situations in which an operation can be invoked, be sure that the preconditions really do describe different situations. If two or more preconditions hold for some combination of values of the abstract state and parameters, then you will not be able to predict which postcondition will hold when the operation completes. In effect, you have ambiguously specified the behavior of the operation. There is a possibility that the implementor will discover what you meant and resolve the ambiguity accordingly. Otherwise, there is a good chance that he will implement the operation in a way that you did not intend. You use the Determinism Criterion to detect and eliminate ambiguities during Class Structuring. The Determinism Criterion can be formally stated as follows, where $P_1, P_2, \ldots, P_L$ are preconditions for an operation:

P_i **and** $P_j \equiv$ **false**

for any i, j in the range [1..L] and $i \neq j$.

The determinism rule holds for the Record Air Temperature operation, as shown below:

P_1 **and** P_2 ≡ SIZE(state_Collection)<6 **and** SIZE(state_Collection)=6
≡ **false**

To see the use of the Determinism Rule, consider the implication of (mistakenly) using "≤" instead of "<" in the first precondition to the Record Air Temperature operation. When the operation is invoked with SIZE(Collection)=6, both preconditions are satisfied and either postcondition could hold:

Postcondition 1: Updated_state_Collection=state_Collection UNION {param_Value}

Postcondition 2: Updated_state_Collection=
(state_Collection− OLDEST(state_Collection))
UNION {param_Value}

In this case, you cannot predict the resulting size of the collection. If Postcondition 2 always holds, the collection will not exceed the bound on its size. However, if Postcondition 1 holds at least some of the time, then the collection will grow, possibly without bound. The ambiguity is revealed by conjoining (i.e., and-ing together) the two preconditions:

P_1 **and** P_2 ≡ SIZE(state_Collection) ≤ 6 **and** SIZE(state_Collection)=6
≡ (SIZE(state_Collection)<6 **or** SIZE(state_Collection)=6)
and SIZE(state_Collection)=6
≡ (SIZE(state_Collection)<6 **and** SIZE(state_Collection)=6)
or SIZE(state_Collection)=6
≡ **false or** SIZE(state_Collection)=6
≡ SIZE(state_Collection)=6

The Determinism rule is a bit stronger than it needs to be. In reality, it is acceptable for two preconditions to overlap in some cases as long as the corresponding effects are the same. A trivial example is obtained by splitting Precondition 1 for the Record Air Temperature operation into two cases:

Precondition 1a: SIZE(state_Collection) ≤ 3
Postcondition 1a: Updated_state_Collection=state_Collection UNION {param_Value}

Precondition 1b: 3 ≤ SIZE(state_Collection)<6
Postcondition 1b: Updated_state_Collection=state_Collection UNION {param_Value}

In this example, preconditions 1a and 1b do not satisfy the Determinism Rule because the state SIZE(state_Collection)=3 satisfies both of them. However, this is not a problem because the effect of the operation is the same in either case. For simplicity, the Determinism Criterion as stated does not address the situation in which overlapping preconditions for an operation produce the same effect or the situation in which the point of overlap is disallowed by the invariants.

In some cases, nondeterministic selection of a postcondition really does not matter. For example, in many cases, an operation can detect multiple errors. If you do not care which error is reported if two or more error conditions exist, you can leave the choice up to the implementor. See Section App.4.5.1 for an example.

5.3.4 Initial State Criterion

From the user's perspective, invariants must hold at all times, including the time before any operations are performed. Therefore, the initial value of the abstract state (if there is one) must satisfy all invariants. The Initial State Criterion can be formally stated as follows, where *S* defines the initial state and $I_1, I_2, \ldots, I_M$ are invariants that are maintained by the class:

S **implies** I_1
S **implies** I_2
. . .
S **implies** I_M.

The Air Temperature Readings Collection Class satisfies the initial state criterion because:

state_Collection={} **implies** SIZE(state_Collection) ≤ 6

5.3.5 Consequence Criterion

You used the Initial State Criterion described in Section 5.3.4 to show that invariants maintained by the class are true initially. If you can also show that no change to the abstract state of an object derived from the class violates these invariants, then you have shown that the invariants are always true. Because only the operations can change the abstract state, this amounts to showing that no operation will cause an invariant to become false. Or, stated positively, you must show that if an operation is invoked when the invariants are true, the invariants will still be true when the operation completes.

To describe the Consequence Criterion, some abbreviations are introduced. Where $I_1, I_2, \ldots, I_M$ are invariants that are maintained by the class and P_i, Q_i represent one of the L precondition-postcondition pairs:

- Let I ≡ I_1 **and** I_2 **and . . . and** I_M.
- Let Updated_I be I with each reference to an attribute of the abstract state state_X replaced with Updated_state_X.

Then the operation satisfies the Consequence Criterion if:

I and P_i **and** Q_i **implies** Updated_I

for all i in the range [1 . . L].

I asserts that all of the invariants are true and refers to the abstract state before the operation takes place. Updated_I refers to the abstract state upon completion of the operation. Often, a postcondition Q_i will not mention each attribute of the abstract state. To prove the above in such cases, you use the convention that anything not mentioned in the postcondition is assumed not to have changed.

Note that the above is **not** the same as:

I and P_1 **and** Q_1 **and . . . and** P_N **and** Q_N **implies** Updated_I

You must show that each precondition-postcondition pair by itself is sufficient to maintain the invariants because upon completion of the operation, you are guaranteed only that one of the postconditions will hold.

The Record Air Temperature operation on the Air Temperature Readings Collection Class satisfies the Consequence Criterion. For the first precondition-postcondition pair:

```
          SIZE(state_Collection) ≤ 6
and       SIZE(state_Collection) < 6
and       Updated_state_Collection = state_Collection UNION {param_Value}
implies   SIZE(Updated_state_Collection) ≤ 6
```

In other words, adding a value to a collection with fewer than six elements results in a collection with no more than six elements. For the second pair:

```
          SIZE(state_Collection) ≤ 6
and       SIZE(state_Collection) = 6
and       Updated_state_Collection =
             state_Collection − OLDEST(state_Collection)  UNION {param_Value}
implies   SIZE(Updated_state_Collection) ≤ 6
```

Or, if the collection has exactly six elements, the software can remove an element and add a new one, and the size of the updated collection will not exceed six.

Notice that including the invariant SIZE(state_Collection) ≤ 6 to the left of the **implies** operator is redundant because each precondition makes a stronger statement about the abstract state. This redundancy is a characteristic of the Record Air Temperature operation; including the invariant is not redundant in general. For example, consider changing the Air Temperature Readings Collection Class so that there are always exactly six entries in the collection and changing the Record Air Temperature operation so that it always removes the oldest value from the collection. Then, the invariant is

SIZE(state_Collection)=6

The single precondition to Record Air Temperature is **true** (because the operation always behaves in the same way), and the single postcondition is:

Updated_state_Collection=
state_Collection −OLDEST(state_Collection) UNION {param_Value}

Now the invariant is necessary because omitting it yields the following, which cannot be proven:

true
and Updated_state_Collection=
state_Collection −OLDEST(state_Collection) UNION {param_Value}
implies SIZE(Updated_Collection)=6

The expression to the left of the **implies** operator makes no mention of the size of the collection, so you cannot infer anything about the size of the collection when the operation completes.

5.3.6 Correctness Analysis

Because class structuring describes static behavior without dynamic behavior, you analyze correctness without considering timing. This approach allows you to show that the controlled variable has the correct value when the operations are invoked at the right time and to meet timing constraints. A complete analysis can be performed during the software architecture design when dynamic behavior is combined with static behavior.

Before you end the class structuring activity, you should convince yourself that you have specified a set of classes and objects that, when invoked, behaves as specified in the requirements. The essence of this analysis is finding a sequence of operations that, when combined with device behavior, will produce the value of a controlled variable required by REQ. You apply this analysis for every *controlled variable in the requirements specification.*

For simplicity, the discussion in this section treats all class operations as functions. When an operation changes the value of a state and a subsequent operation uses the value, that value is treated as the function result of the "set" operation and the input to the "get" operation. For example, the Set_Report and Get_Next_Page operations of the ASCII report data abstraction class use a state variable to describe behavior.

Correctness can be shown when, for every behavior defined by a value function in REQ, there exists a composition of operations (combined with device behavior) equal to that value function. If this analysis is done for class structuring, it will reduce the probability of rework necessitated by software architecture design when process logic is updated with invocations to these operations. This composition of functions correlates to the stimulus/response threads evaluated during process structuring.

One way to begin the analysis is to find a composition of functions that is at least well defined and satisfies the preconditions specified for each function. Using the abbreviations VF for value function and OP for operation, the goal can be expressed informally as:

CON.VF (mon_V) =
OUT.VF (out_V.OP (... ~ con_V.OP(... ~ mon_V.OP(...in_V.OP (IN.VF (mon_V))...)...)...))

The following example is an informal walkthrough to illustrate how correctness analysis would begin. The example uses the case study where an SOS_Report is produced every 60 seconds when in mode_SOS (see Section App.2.11.1):

Assume:
[mon_Time mod 0] = 60 and
INMODE (mode_SOS)

Show that:
There exists a composition of operations for the value of con_Report.ASCII_Report, ASCII(term_SOS_Report).

The term_SOS_Report (see Section App.2.6) consists of only one element, mon_Buoy_Location, so begin with the input device that provides mon_Buoy_Location (see Section App.2.10.1):

IN_for_mon_Buoy_Location:
in_Omega_System_Input = mon_Buoy_Location + mon_Omega_Error ($\pm$0.4 km)

Using the notation introduced earlier:

in_Omega_System_Input =
IN_for_mon_Buoy_Location.VF(mon_Buoy_Location, mon_Omega_Error)

The input variable that the software uses is in_Omega_System_Input. You can now focus on class specifications derived from that input variable or device. You most likely need to approximate mon_Buoy_Location:

OMEGA_NAVIGATION_SYSTEM DEVICE INTERFACE CLASS:
get_Omega_Input (see App.4.2.1)

BUOY_LOCATION COMPUTATION CLASS:
estimate_Buoy_Location (Omega_System_Input) (see App.4.10.1)

Looking at the postcondition of get_Omega_Input, the value returned depends on the value of in_Omega_System_Input. To continue using functional notation, use a parameter instead of referring to the input variable to get:

~mon_Buoy_Location =
estimate_Buoy_Location(get_Omega_Input
(IN_for_mon_Buoy_Location.VF(mon_Buoy_Location, mon_Omega_Error)))

There are now many candidate operations because ~mon_Buoy_Location is used several ways. Because the requirement under consideration is to generate a report, concentrate on classes derived from the con_Report variable:

SOS_REPORT DATA ABSTRACTION CLASS:
ASCII_Format (Set_Latitude (Latitude), Set_Longitude (Longitude)) (see App.4.7.1 and App.4.7.2)

In fact, the latitude and longitude must be from the ~mon_Buoy_Location, which you already have. Because the ASCII_Format operation approximates the value of con_Report_ASCII, look for

operations that can generate an output variable from ~con_Report_ASCII. Treat the state variable as a function result and parameter to continue using the function notation for this walkthrough:

ASCII_REPORT DATA ABSTRACTION CLASS:
Get_Next_Page (Set_Report (~con_Report_ASCII)) (see App.4.5.1 and App.4.5.2)

Because you are dealing with a one-page report, you do not have to deal with iteration, and one invocation of Get_Next_Page is sufficient. Also, because there is at least one page, the precondition state_Pages_Remaining is true. Now you only need to apply the value function for the output device (see Section App.2.11.2) and compose it with the monitored variable approximation above:

con_Report.ASCII_Report =
Out_for_con_Report (Get_Next_Page (Set_Report (ASCII_Format (Set_Latitude (~mon_Buoy_Location.Latitude), Set_Longitude (~mon_Buoy_Location.Longitude)))))

where

~mon_Buoy_Location =
estimate_Buoy_Location(get_Omega_Input
(IN_for_mon_Buoy_Location.VF(mon_Buoy_Location, mon_Omega_Error)))

Of course, this only shows a possible composition of operations. By using a strong typing system or showing that each operation's preconditions hold, you can show that the composition is well defined. But correctness requires that the composition equal the value function given the assumptions; i.e., that the composition of functions above simplifies to:

con_Report.ASCII_Report = ASCII(term_SOS_Report)

To finish the analysis, each of the value functions and operations in the composition is replaced with the appropriate expressions that describe the value returned. Then, the expression is simplified to determine whether it equals the expression used to describe the required behavior.

5.3.7 ERROR ANALYSIS

When error and delay functions are independent (see Section 3.3), error analysis can be performed before merging the dynamic view of the system in software architecture design. Each of the function compositions described in Section 5.3.6 can be evaluated for the worst case error.

Each of the operations must be allocated some allowed error in the form of a constant value or function of the operation inputs and state. This is not necessary for discrete valued objects and their classes.

Beginning with the input device, maximum error propagation can be calculated as each function is applied until the value of the controlled variable is calculated. If this propagated error is less than the tolerable error in the CoRE specification for that controlled variable, then the error tolerance has been met.

5.4 FUTURE WORK

The following topics should be considered in future versions of the class structuring guidelines:

1. The evaluation criteria discussed in Section 5.3 should consider computational error.

2. The evaluation criteria should allow specialization classes to inherit characteristics informally proven about the generalization parent (generalization and specialization classes are described in Section 9.6 of the ADARTS Guidebook).

3. There should be additional guidance for writing invariants, preconditions, and postconditions that relate the abstract state of a class to the requirements traced to the class (e.g., assumptions relating the abstract state of the Air_Temperature_Readings Collection Class in the HAS Buoy case study to term_Averaged_Air_Temperature).

6. SOFTWARE ARCHITECTURE DESIGN

The software architecture design activity of the ADARTS method does not use requirements artifacts as input. Therefore, the essence of the heuristics is relatively immune to change. However, the approach presented in this document suggests that CoRE's precision can continue to be exploited as you apply ADARTS heuristics. This section highlights the advantages of using a precise specification of behavior during software architecture design. The guidelines in this section are optional; you do not have to follow them to produce an ADARTS design from CoRE requirements. However, you should strongly consider following these guidelines if you attempted to maintain CoRE's level of precision in process and class structuring. Use this section with Section 10 of the ADARTS Guidebook.

The entrance criteria for software architecture design remain the same. You must have the process architecture diagram, process behavior specifications, class specifications, and dependency graph work products. The following sections describe the software architecture design activities in terms of the more precisely defined work products:

- Merging the dynamic and static views of the design into the software architecture diagram and updating the process logic (see Section 6.1)
- Identifying the need for resource monitors (see Section 6.2)
- Evaluating the software architecture design (see Section 6.3)
- If delay and error were specified as mutually dependent (see Section 3.3), analyzing delay and error constraints in software architecture design (see Section 6.4)

6.1 MERGING DYNAMIC AND STATIC VIEWS

Use this section with Section 10.4 of the ADARTS Guidebook.

The key guideline in merging the dynamic and static views of the system is to evaluate the requirements traceability in the process behavior specifications and class specifications.

A general procedure is presented in Section 6.1.1 followed by an example in Section 6.1.2.

6.1.1 General Procedure

For each process behavior specification, examine the process logic and traceability to find the operations on objects that match each response in a stimulus/response pair. Update the process logic to invoke the operation with appropriate parameters.

Ensure that the preconditions for the operation are met when the operation is invoked. Several stimuli may respond to the same event under different conditions. You must match the operation with the response in such a way that the precondition is true.

Each time an operation is used to respond to a stimulus, add a dependency from the process to the operation of that object (if it does not already exist). This dependency is reflected in the software architecture diagram, which is used later to evaluate the need for resource monitors.

6.1.2 Example of Updating Process Logic

Consider the process logic of the Generate_Periodic_Reports process in Section App.3.4.4. Using the requirements traceability and naming conventions, each expression is replaced with invocations of operations found in class specifications. Table 15 shows an example of a portion of the process logic, how it is updated, and the classes used from the class specifications.

Table 15. Example of Updating Process Logic

Process Logic	Updated Process Logic	Class Specification
if (System_Mode = "mode_SOS") then	if Current_Mode = Emergency then	System_Mode State Transition Class App.4.9
Report.Report_Type <--"SOS_Report"	Report.Report_Type <--"SOS_Report"	
SOS_Report <-- read Buoy_Location data store	read latitude (SOS_Report.latitude) read longitude (SOS_Report.longitude)	Buoy_Location Data Abstraction Class App.4.6
Report.ASCII_Report<-- ASCII(SOS_Report)	Set_Latitude (SOS_Report.latitude) Set_Longitude (SOS_Report.longitude) Report.ASCII_Report <--ASCII_Format	SOS_Report Data Abstraction Class App.4.7
send Report to Report_Queue	send Report to Report_Queue	

The software architecture diagram is updated appropriately. A partial diagram showing the dependencies that are added based on the updated process logic appears in Figure 42.

6.2 RESOURCE MONITORS

Use this section with Section 10.5 of the ADARTS Guidebook.

There is no significant change in dealing with multiple processes accessing the same object. The same procedure is followed to ensure access synchronization and data protection:

- Add a resource monitor class with the same operations as the original class. Use the same class behavior specification modified slightly to indicate synchronous access.
- Update the dependency graph to show that the object depends on the new resource monitor class.

6.3 EVALUATION CRITERIA

Use this section with Section 10.7 of the ADARTS Guidebook.

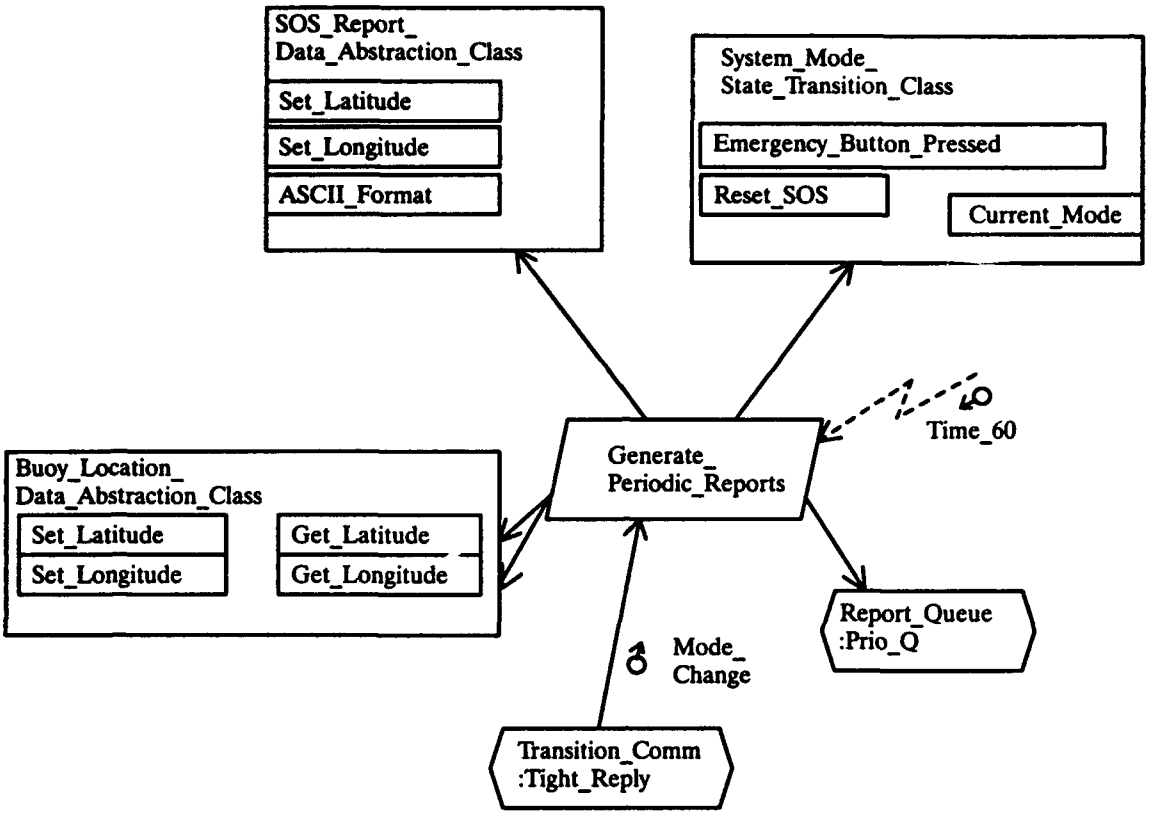


Figure 42. Partial Software Architecture Diagram Illustration

With the added precision of CoRE and the formalisms introduced in this report, additional evaluation criteria can be introduced:

- The preconditions for every operation must be true each time the operation is invoked by a process. For example, in the example shown in Table 15, invoking ASCII_Format can result in an error if NOT(location_defined). But a quick check shows that invoking Set_Latitude and Set_Longitude ensures that location_defined is true. Therefore, invoking ASCII_Format cannot result in an error.

- The timing constraints of each process should be reasonable when considering the operations that must be performed. This may include assigning processing time to individual operations. Then, each process can be evaluated to see if each response can be performed within the allotted execution time. A more direct analysis might include evaluating each stimulus/response thread using the cumulative time to execute all operations in a response instead of the estimated execution time.

- The additional delay introduced by resource monitors does not cause a violation of the timing constraints.

- An informal proof of correctness can again be accomplished using the same procedure outlined in Section 4.5.3.

- A more detailed and complete error analysis can be performed in software architecture design by evaluating each stimulus/response thread. However, if a complete analysis performed in

class structuring did not reveal any problems, you are unlikely to find any problems in software architecture design. More likely, you may have deferred some strict constraints on the precision of operations until a more complete analysis could be done of how the operations would be used (see Section 6.4).

6.4 RELATING DELAY AND ERROR

As discussed in Section 3.3, delay and error can be mutually dependent. Because delay is a concern in process structuring and error a concern in class structuring, it is usually easier to design software that meets delay and error constraints separately. However, if the constraints are tight, you may have to take the relationship between delay and error into account. For example, if you cannot meet a delay constraint, it may be possible to allow more time for a stimulus-response thread to complete by restricting the error introduced by the classes participating in the thread. This section contains an example of dealing with the mutual dependency between delay and error.

The following example of a digital speedometer illustrates how you can analyze the delay and loss of precision introduced by software components and hardware devices participating in a stimulus-response thread and can ensure that your design does not exceed tolerance for inaccuracy or bounds on delay. The approach illustrated in this section makes three important simplifying assumptions:

1. This approach assumes that the delay associated with each component is constant (i.e., that each component always takes the same amount of time to transform inputs to outputs). You can use this approach to perform a worst case analysis of your design. If the outputs are within range, assuming worst case performance, they will be within range in all cases. However, failure to pass a worst case analysis does not imply anything about average-case performance. It is possible for a design to fail under a low-probability scenario and still work often enough to meet the requirements.

2. Each component in the design transforms one or more inputs to one or more outputs. In the general case, the error introduced by a transformation will depend on the input value. This approach assumes that the inaccuracy introduced by each input/output device and process is constant. As with the previous simplifying assumption, the implication is that this approach can be used for a worst case analysis but may not necessarily indicate how the design will perform in the average case.

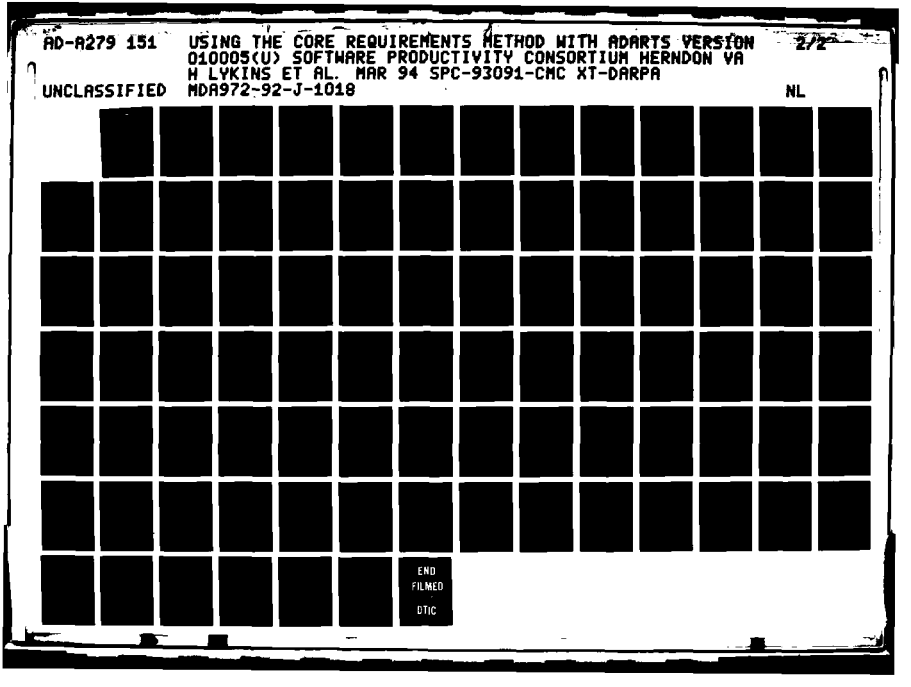
3. This approach does not consider the error introduced by arithmetic operations on floating-point numbers. See Knuth (1981) for a discussion of the accuracy of floating-point arithmetic.

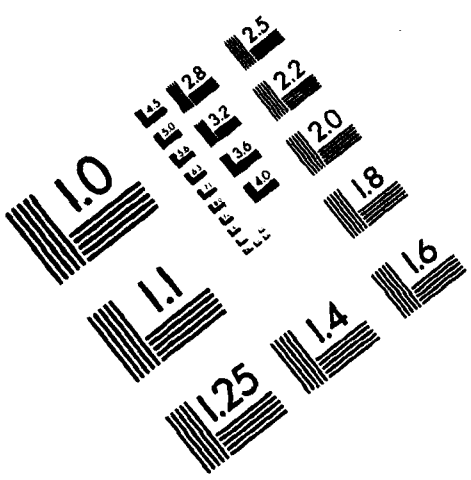
6.4.1 Examples of Requirements

The environmental variables are defined in Table 16:

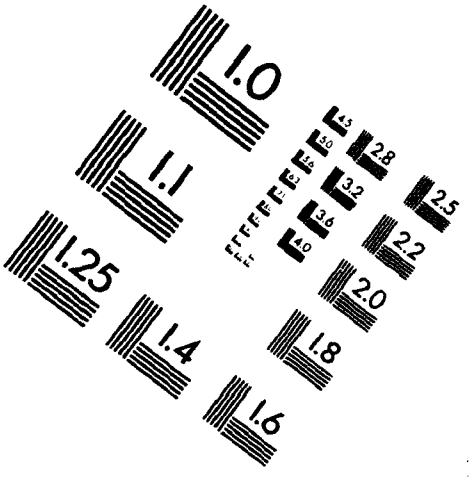
Table 16. Environmental Variables

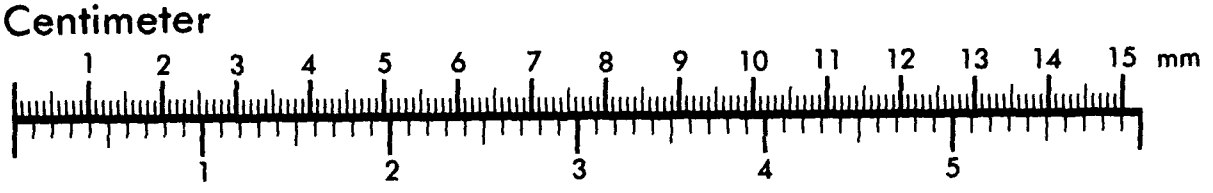
Name	Type	Values	Physical Interpretation
mon_Actual_Speed	Real	0 to 120 mph	The actual speed of the automobile
con_Displayed_Speed	Real	0 to 120 mph	The value displayed on the driver's console.

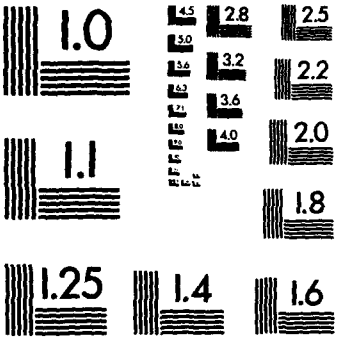

AD-A279 151
USING THE CORE REQUIREMENTS METHOD WITH ADARTS VERSION 010005(U) SOFTWARE PRODUCTIVITY CONSORTIUM HERNDON VA
H LYKINS ET AL. MAR 94 SPC-93091-CMC XT-DARPA
2/2
UNCLASSIFIED
MDA972-92-J-1018
NL
END
FILMED
DTIC

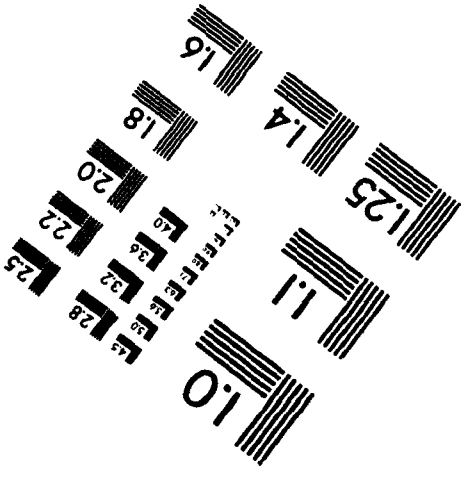


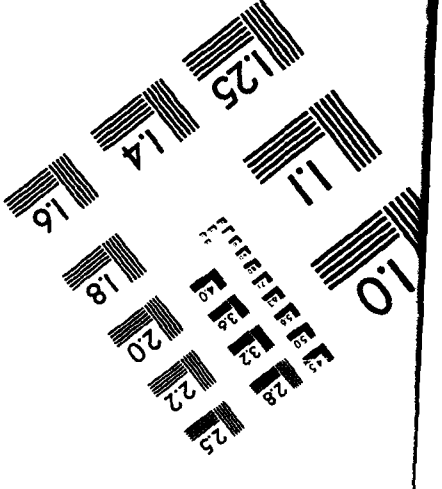
Association for Information and Image Management
1100 Wayne Avenue, Suite 1100
Silver Spring, Maryland 20910
301/587-8202











MANUFACTURED TO AIIM STANDARDS
BY APPLIED IMAGE, INC.

The variable con_Displayed_Speed is an integer; the display device is not capable of displaying decimal values.

The requirements for the speedometer are:

Ideal REQ Relation: con_Displayed_Speed = mon_Actual_Speed
Tolerance: The displayed speed must not differ from the actual speed by more than 1 mph.
NAT Relation: Acceleration and deceleration cannot exceed 5 mph per second:

$$\left|\frac{d}{dt}\text{mon\_Actual\_Speed}\right| \leq \frac{5\ \text{mph}}{\text{sec}}$$

The above NAT relation assumes that the reading of the speedometer need not be maintained in a crash situation, i.e., when the rate of deceleration is greater than 5 mph per second.

6.4.2 Example of Design and Informal Evaluation

The process architecture and input/output devices for the digital speedometer appear in Figure 43. For simplicity, only the processes are shown on the software architecture diagram; objects are omitted. In this simple example, each component takes a single input and produces a single output. An input sensor measures the actual speed of the automobile, producing an input data item that is converted by a process into an internal approximation of the actual speed. Another process truncates the approximation into an integer, which a third process outputs to a display device. The display device then formats the integer and causes it to appear on the driver's console. Interprocess communications and communications between processes and input/output devices in Figure 43 are annotated with the value being communicated.

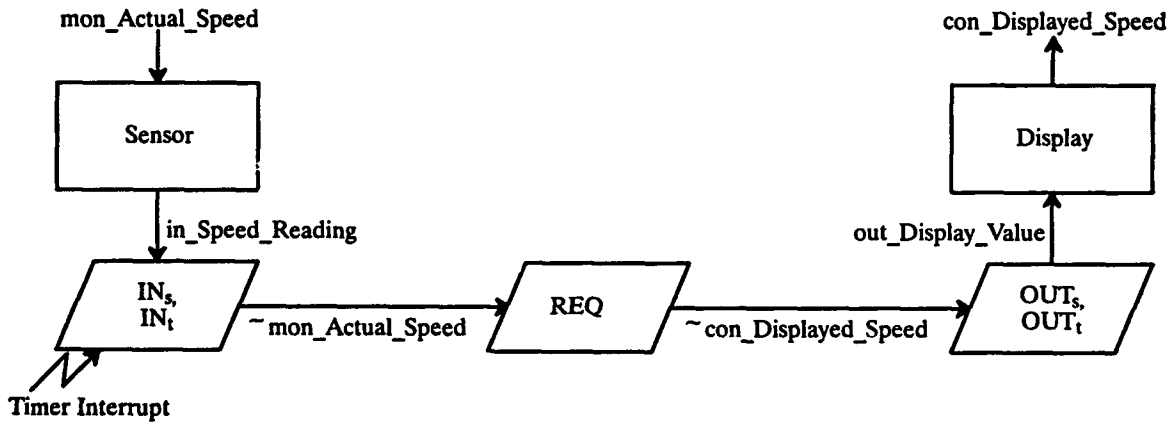


Figure 43. Process Structure for Digital Speedometer

The following describes the behavior of each component. Associated with each component is the ideal relationship between its input and its output, an upper bound on error, and an upper bound on the amount of time required for the component to produce an output given an input.

Sensor:

IN Relation:	in_Speed_Reading = mon_Actual_Speed
Error:	within 0.1 mph
Delay:	0.01 sec

IN_s, IN_t process:

Behavior: ~mon_Actual_Speed$_{Approx}$ = in_Speed_Reading
Error: No loss of accuracy
Delay: 0.02 sec

REQ Process:

Behavior: ~con_Displayed_Speed = integer(~mon_Actual_Speed + 0.5mph)
Error: The "integer" operator introduces a loss of accuracy of up to 1 unit of measurement.
Delay: 0.1 second

OUT_s, OUT_t process:

Behavior: out_Displayed_Value = ~con_Displayed_Speed
Error: No loss of accuracy
Delay: 0.02 sec

Display Device:

OUT Relation con_Displayed_Speed = out_Displayed_Value
Error: No loss of accuracy
Delay: 0.01 sec

To ensure that the design meets timing and accuracy requirements, use the above relationships to derive a relationship between con_Displayed_Speed and mon_Actual_Speed. This relationship will depend upon how the software and input/output devices generate con_Displayed_Speed from mon_Actual_Speed. Then compare this relationship with the one in the requirements specification. To derive this relationship, begin with the sensor's translation of mon_Actual_Speed to in_Speed_Reading and work to the end result. First, consider only loss of accuracy introduced by the devices or computations within processes. Then consider additional loss of accuracy due to delay.

1. The maximum loss of accuracy introduced by the input sensor is 0.1 mph. Therefore, in_Speed_Reading is within 0.1 mph of mon_Actual_Speed.

2. The first process in the thread introduces no loss of accuracy. Combining its behavior with that of the input sensor implies that ~mon_Actual_Speed is also within 0.1 mph of mon_Actual_Speed.

3. Because the REQ process rounds its input to the nearest integer, it introduces an error of, at most, 0.5 mph, implying that ~con_Displayed_Speed$_x$ is within 0.6 mph of ~mon_Actual_Speed.

4. Neither the "OUT_S,OUT_T" nor the display device introduce any inaccuracy. Therefore, con_Displayed_Speed is within 0.6 mph of mon_Actual_Speed.

The analysis thus far shows that the process structure meets the timing and accuracy requirements. However, you must also consider that each component takes a nonzero period of time to operate, during which time mon_Actual_Speed can be changing at up to 5 mph per second. This can cause an additional discrepancy between the current values of con_Displayed_Speed and mon_Actual_Speed.

1. During the 0.01 second required by the sensor to produce its input data item, mon_Actual_Speed can change by, at most, 0.05 mph, implying that in_Speed_Reading is really within 0.15 mph of mon_Actual_Speed.

2. The process "IN_S,IN_t" requires 0.02 second to read the input data item. During this time, mon_Actual_Speed can change by, at most, 0.1 mph, implying that ~mon_Actual_Speed is actually within 0.25 mph of mon_Actual_Speed.

3. During the 0.1 second required by the REQ process to round ~mon_Actual_Speed, it is possible for mon_Actual_Speed to change by another 0.5 mph. Adding the inaccuracy associated with the "integer" function implies that ~con_Displayed_Speed is within 1.25 mph of mon_Actual_Speed. Thus, the design exceeds the accuracy requirement, and the analysis is not finished.

4. During the 0.02 second required by the "OUT_S,OUT_T" process to send the output data item to the environment, mon_Actual_Speed can change by an additional 0.1 mph, implying that out_Displayed_Value is within 1.35 mph of mon_Actual_Speed.

5. Finally, the Display Device requires 0.01 second to update the driver display, during which time mon_Actual_Speed can change by another 0.05 mph. Thus, it is possible for con_Displayed_Speed to differ from mon_Actual_Speed by as much as 1.40 mph, a total of 0.40 mph beyond the maximum allowable tolerance specified in the requirements.

The first analysis, which disregarded inaccuracy introduced by delay, implied that the design met the requirements. However, this analysis, which took delay into consideration, shows that the design fails to meet accuracy requirements. At this point, change the design, change assumptions made about the timing and/or accuracy of one or more components, or try to convince the customer to change the requirements. In this example, take the second option.

The most time-consuming component in the process structure is the "REQ" process, which takes up to five times as long as any other component to translate its input into an output. Assume that a change in data structures and algorithms will reduce the amount of time required by this process from 0.1 second to 0.01 second. The updated description of the "REQ" process is:

REQ Process:

Behavior:	~con_Displayed_Speed = integer(~mon_Actual_Speed + 0.5 mph)
Error:	The "integer" operator introduces a loss of accuracy of up to 1 unit of measurement.
Delay:	0.01 second

The updated analysis is:

1. The variable in_Speed_Reading is within 0.15 mph of mon_Actual_Speed, as before.

2. The variable ~mon_Actual_Speed is actually within 0.25 mph of ~mon_Actual_Speed, as before.

3. Because the REQ process now needs only 0.01 second, the maximum change in ~mon_Actual_Speed is 0.05 mph, implying that ~con_Displayed_Speed will be within 0.25 + 0.55 = 0.8 mph of mon_Actual_Speed.

4. As before, ~mon_Actual_Speed can change by an additional 0.1 mph during execution of the "OUT_S,OUT_T" process, implying that out_Displayed_Value is within 0.9 mph of mon_Actual_Speed.

5. The additional delay of 0.01 second imposed by the Display Device will allow mon_Actual_Speed to change by another 0.05 mph, yielding a maximum discrepancy of 0.95 mph between con_Displayed_Speed and mon_Actual_Speed. This time, the result is well within the required bounds on error.

By revising your assumption about the timing of the REQ process, you have managed to produce a design that meets the accuracy requirements, according to the above analysis. However, there remains one simplifying assumption that must be discarded before you can have confidence in the process structure. Neither of the analyses considered the frequency of the timer interrupt that triggers the "IN_s,IN_t" process, assuming instead that the sensor is polled constantly. To complete the analysis, consider the additional delay resulting from a finite polling frequency and the effect of this delay on accuracy.

Assuming constant polling, the discrepancy between con_Displayed_Speed and mon_Actual_Speed is 0.95 mph. If the delay introduced by periodic polling does not exceed 0.55 mph, the design will still meet the requirements. To determine how much time can elapse between successive polls of the input sensor, recall the NAT relation:

$$\left|\frac{d}{dt}\text{mon\_Actual\_Speed}\right| \leq \frac{5\ \text{mph}}{\text{sec}}$$

or

$$dt = \frac{0.2\ \text{sec}}{\text{mph}} \times d(\text{mon\_Actual\_Speed})$$

If you maximize dt subject to the condition that d(mon_Actual_Speed) $\leq$ 0.05 mph, the result is:

$$dt \leq \frac{0.2\ \text{sec}}{\text{mph}} \times 0.05\ \text{mph}$$

or

$$dt \leq 0.01\ \text{sec}$$

Thus, a polling rate of at least 100 times per second will satisfy the accuracy bounds on con_Displayed_Speed.

6.5 FUTURE WORK

Most of the activity of merging static and dynamic views of the system should be mechanical. Although the precision of process structuring and class structuring work products simplifies the merging activity, there are still disconnects. This apparently results from using a different set of criteria when defining responses in the stimulus response threads and operations in the class specifications.

Using this new level of precision, heuristics should be defined for defining responses and operations that make the software architecture activity of updating process logic very mechanical (potentially automatable).

APPENDIX: HAS BUOY CASE STUDY

This section contains the HAS Buoy case study, including a description of the problem, a CoRE requirements specification, and ADARTS process and class structures built from the CoRE specification. Experience obtained from this case study was used to identify and validate guidance provided in the main portion of this report.

The CoRE work products contained in this section were developed using **teamwork**/RT and tailored according to the most recent version of CoRE. The ADARTS work products contained in this section were developed using **teamwork**/Ada and the guidance contained in Kirk and Wild (1992). In most cases, the data dictionary entries have been defined using the syntax supported by **teamwork**/RT's checking facility (refer to *Teamwork/SA and teamwork/RT User's Guide* [Cadre Technologies, Inc. 1990]). Parts of the **teamwork** model that are specific to **teamwork**, such as references to database identifiers of **teamwork** objects, have been omitted.

Section App.1 contains the HAS Buoy problem statement. Section App.2 contains the CoRE specification developed for the HAS Buoy. Section App.3 contains the ADARTS process structure that was derived from the CoRE specification. Section App.4 contains the ADARTS class structure that was derived from the CoRE specification.

APP.1 HAS BUOY PROBLEM STATEMENT

This section contains the HAS Buoy problem statement. This problem statement was adapted from *Software Engineering Principles* (Naval Research Laboratory 1980). This problem statement has been modified from its use in previous case studies, such as the ADARTS Guidebook.

App.1.1 Introduction

The Navy intends to deploy HAS buoys to provide navigation and weather data to air and ship traffic at sea. The buoys will collect wind, temperature, and location data and will periodically broadcast summaries. Passing vessels will be able to request more detailed information. In addition, HAS buoys will be deployed in the event of accidents at sea to aid sea search operations.

Each HAS buoy will contain a small computer and a number of devices for interacting with its environment. Section App.1.7 specifies the resources that are available to the HAS buoy, including:

- Wind sensors for determining wind magnitude and direction (see Section App.1.7.1)
- Temperature sensors for determining air and water temperature (see Section App.1.7.2)
- A radio receiver and radio transmitter for communicating with passing vessels (see Section App.1.7.3)

- A panel containing an emergency button and a red light (see Section App.1.7.4)
- An Omega receiver for obtaining location information from the Omega navigation system (see Section App.1.7.5)

App.1.2 Software Requirements

The software for the HAS buoy must satisfy the following requirements:

- Maintain current wind and temperature information by monitoring sensors regularly and averaging readings.
- Calculate location via the Omega navigation system.
- Broadcast wind and temperature information every 60 seconds.
- Broadcast more detailed reports in response to requests from passing vessels. The detailed reports contain buoy location information in addition to the information contained in the wind and temperature reports.
- Broadcast weather history information upon request. These weather history reports consist of all wind and temperature reports produced in the last 48 hours.
- Broadcast an SOS signal in place of the ordinary wind and temperature report after a sailor presses the emergency button. SOS signals, including buoy location information, should be broadcast periodically (every 60 seconds) until a vessel sends a reset signal.
- Cause the buoy's red light to begin flashing and stop flashing in response to requests from passing vessels.
- Accept location correction information via the radio receiver from passing vessels. The software must use this information to modify its calculation of location based on Omega information.

App.1.3 Reports

The contents of the reports are as follows:

- Wind and temperature report contains the averages of each of the following over the previous 60 seconds: air temperature, water temperature, and wind magnitude and direction.
- SOS report identifies the location of the buoy.
- Detailed report contains the buoy location plus the averages of each of the following over the previous 60 seconds: air temperature, water temperature, and wind magnitude and direction.
- Weather history report contains all wind and temperature reports broadcast over the last 48 hours.

Each report must be converted to ASCII characters and transmitted in RAINFORM format. The ASCII form of each field of a report will be as identified in Table 17.

Table 17. Report Notation

Report Field	ASCII Notation
Temperature	"sddd"
Buoy location	"ddd°dd'dd.dd"bddd°dd'dd.dd""
Wind direction	"ddd"
Wind magnitude	"ddd"

where:

- "s" = sign (blank space if positive, "−" if negative).
- "b" = blank space.
- "d" = single digit (leading zeros must always be used, and numerical values must be rounded upward).
- Other characters represent literals.

App.1.4 Software Timing Requirements

In order to maintain accurate information, readings must be taken from the sensing devices at the following fixed intervals:

Temperature sensors:	every 10 seconds
Wind sensors:	every 30 seconds

App.1.5 Priorities

Reports will be broadcast over the radio transmitter according to the following priority ranking:

SOS	1	highest
Wind and temperature	1	
Detailed (ship and airplane)	2	
Weather history	3	lowest

Transmission of lower priority reports will be interrupted when higher priority reports become ready to be transmitted. Transmission of interrupted reports must be completed upon transmission of higher priority reports.

App.1.6 Error Detection

The software will respond to erroneous input from the set of wind sensors by ignoring such data. Sensor input is erroneous when opposing sensors provide conflicting information (see Section App.1.7.1).

App.1.7 HAS Buoy Device Specifications

This section describes the interfaces to the devices with which the HAS Buoy software system interacts.

App.1.7.1 Wind Sensors

There are four wind sensors, each of which measures the force of the wind from its respective direction (i.e., due north, south, east, or west). Table 18 specifies the relevant information for each sensor.

Table 18. Wind Sensor Specifications

Device	Description	Range/Units	Size	Address
North	Wind magnitude in due north direction	0 to 255 knots	8-bit unsigned integer	Port C1
South	Wind magnitude in due south direction	0 to 255 knots	8-bit unsigned integer	Port C2
East	Wind magnitude in due east direction	0 to 255 knots	8-bit unsigned integer	Port C3
West	Wind magnitude in due west direction	0 to 255 knots	8-bit unsigned integer	Port C4

Note that any force detected by a wind sensor means that the opposing sensor should register no wind (e.g., if the north sensor detects wind in the north direction, the south sensor should detect zero wind).

The wind sensors are passive devices that may be sampled at any time. It takes, at most, 1 second for the wind sensors to detect a change in wind magnitude and/or direction.

App.1.7.2 Temperature Sensors

There are two independent air temperature sensors and two independent water temperature sensors that provide measurements of air and water temperature in degrees centigrade. Table 19 specifies the relevant information for each sensor.

Table 19. Temperature Sensor Specifications

Device	Description	Range/Units	Size	Address
Air1	Air temperature ten feet above the water surface	−128°C to 127°C	8-bit two's-complement integer	Port B1
Air2	Air temperature ten feet above the water surface	−128°C to 127°C	8-bit two's-complement integer	Port B2
Water1	Water temperature four feet below the water surface	−128°C to 127°C	8-bit two's-complement integer	Port A1
Water2	Water temperature four feet below the water surface	−128°C to 127°C	8-bit two's-complement integer	Port A2

The temperature sensors are passive devices that may be sampled at any time. It takes, at most, 1 second for the temperature sensors to reflect a change in air or water temperature.

App.1.7.3 Radio

The radio receiver is capable of receiving 3-byte message packets. Message packets are made up of the following two components:

- Message Type:

  Byte 1: 16#01# = request to turn on the buoy's red light
  16#02# = request to turn off the buoy's red light
  16#03# = request for a weather history report
  16#04# = request for a detailed report
  16#05# = request to terminate transmission of SOS signals
  16#06# = submittal of location correction data (see Supplemental Data)
  others = none

- Supplemental Data:

  Bytes 2 and 3: if Byte 1 indicates submittal of location correction data then:
  Byte 2: 8-bit two's complement integer indicating Omega system error correction for latitude calculation in kilometers
  Byte 3: 8-bit two's complement integer indicating Omega system error correction for longitude calculation in kilometers
  otherwise this byte is unused.

The radio receiver is an active device that sets Bytes 1, 2, and 3 according to message type each time an incoming message is detected. Acknowledgment of messages received by the software is performed by resetting Byte 1. It takes, at most, 6.0 seconds for the radio receiver to identify and capture an incoming message. Table 20 provides additional information about the radio receiver.

The radio transmitter is capable of transmitting 512-byte message packets. Message packets are made up of the following three components:

- Packet Type:

  Byte 1: 2#10000001# means bytes 3 to 512 contain a page of an SOS report
  2#10000010# means bytes 3 to 512 contain a page of a wind and temperature report
  2#10000011# means bytes 3 to 512 contain a page of a detailed report
  2#10000100# means bytes 3 to 512 contain a page of a weather history report
  2#0xxxxxx# means no message should be transmitted

- Packet Identifier:

  Byte 2: Bits 0 to 3: 4-bits range 1 to 16 representing total number of pages in report
  Bits 4 to 7: 4-bits range 1 to 16 representing number of page being transmitted

- Packet Buffer:

  Bytes 3 to 512: Report page represented by 8-bit ASCII characters
  End of report represented by 16#FF#

The radio transmitter is an active device that broadcasts a packet each time the most significant bit of Byte 1 is set. Upon completion of a broadcast, the bit is automatically reset. It takes, at most, 10.0 seconds to transmit each packet. Table 20 provides additional information about the radio transmitter.

Table 20. Radio Device Specification

Device	Description	Range/Units	Size	Address
Radio Transmitter	Broadcasts messages over a preset radio frequency	See above	512 bytes	Port G
Radio Receiver	Receives messages from a preset radio frequency	See above	2 bytes	Port F

App.1.7.4 Buoy Panel

The light on the buoy panel is a passive device that is manipulated by setting or resetting the controller bit as specified in Table 21. It takes, at most, 0.5 second for the light to turn on or off after a request has been made.

The emergency button is an active device that indicates the status of the button as specified in Table 21. It takes, at most, 0.1 second for this bit to detect a change in status of the emergency button.

Table 21. Buoy Panel Device Specification

Device	Description	Range/Units	Size	Address
Light Switch	Controls the operation of the red light on the panel	0 (Off) 1 (On)	Most significant bit of 8-bit byte	Port H
Emergency Button	Indicates the status of the emergency button on the panel	0 (Released) 1 (Pressed)	Most significant bit of 8-bit byte	Port E

App.1.7.5 Omega Navigation System

The Omega navigation system periodically (every 30 seconds) broadcasts location information that is obtained by the buoy's on-board Omega system receiver within 10 seconds. The receiver is a passive device, updated periodically, that indicates buoy location using the following representation:

Bytes 1 and 2: Degrees latitude range 0 to 65,535, 16-bit unsigned integer
Byte 3: Minutes latitude range 0 to 255, 8-bit unsigned integer
Byte 4: Whole seconds latitude range 0 to 255, 8-bit unsigned integer
Byte 5: 1/100th seconds latitude range 0 to 255, 8-bit unsigned integer
Bytes 6 and 7: Degrees longitude range 0 to 65,535, 16-bit unsigned integer
Byte 8: Minutes longitude range 0 to 255, 8-bit unsigned integer
Byte 9: Whole seconds longitude range 0 to 255, 8-bit unsigned integer
Byte 10: 1/100th seconds longitude range 0 to 255, 8-bit unsigned integer

Table 22 provides additional device information related to the Omega system.

Table 22. Omega Device Specification

Device	Description	Range/Units	Size	Address
Omega	Provides buoy location information as indicated by the Omega navigation system	See above	10 bytes	Port D

APP.2 CoRE REQUIREMENTS SPECIFICATION

This section contains the CoRE requirements specification developed for the HAS Buoy problem. This section is decomposed into a number of subsections, each of which contains a particular kind of CoRE requirements artifact:

- Section App.2.1 contains the information model.
- Section App.2.2 contains the context diagram.
- Section App.2.3 contains the dependency graph.
- Section App.2.4 contains definitions of monitored and controlled variables.
- Section App.2.5 contains definitions of input and output variables.
- Section App.2.6 contains definitions of CoRE events and terms.
- Sections App.2.7 through App.2.13 contain the CoRE artifacts relevant to each of the CoRE classes identified on the dependency graph, including:
  - Mode machines
  - REQ relations, including relevant behavior
  - IN and OUT relations, including the inverse of each value function (IN', OUT')
  - NAT relations
- Section App.2.14 contains the remaining **teamwork** data dictionary entries referenced from data dictionary entries used to define CoRE artifacts.

The conventional use of **teamwork** was tailored in the following ways:

- Data dictionary entries have been created to define terms used in the definitions of other data dictionary entries. Those functions that are defined by data dictionary entries are named with a combination of upper and lower case letters. Those for which a commonly understood definition of the function is assumed (e.g., COS, SIN, ROUND, SQRT) are named with all upper case letters.
- The CoRE convention of labeling requirements artifacts with certain prefixes (e.g., "mon_", "in_," "term_") has been adhered to; however, the convention has been extended to include the following:
  - "NAT_": a NAT relation.
  - "behavior_of_": detailed behavior description (scheduling constraints) associated with controlled variables and their REQ relations.
  - "timing_": timing and error information associated with devices specified by IN and OUT relations. Devices are classified as one of the following (ADARTS terminology for devices was used):

-- ***Passive***: The software may sample an input or produce an output at any time, without synchronization.

-- ***Active***: Interrupt mechanisms are required to synchronize inputs and outputs.

-- ***Periodic***: Input variables are updated periodically, and output variables are processed periodically by the device.

- Detailed behavior descriptions (scheduling constraints) associated with controlled variables and their REQ relations (see data dictionary entries [DDEs] prefixed by "behavior_of_") have been extended to include identification of relevant events. The frequency profile of each event, including minimum, expected, and maximum intervals, is included in the DDEs for the events rather than the detailed behavior descriptions.

- Because most of the IN, REQ, and OUT relations in this case study are not dependent upon the mode of the system (see Mode_Class_for_mode_System_Mode), the notation recommended by the CoRE Guidebook for defining these relations was not particularly useful. Therefore, the notation was tailored for this use. With one exception (REQ_Relation_for_con_Report), relations in this case study will take one of the following two forms:

Condition	Variable
C_1	V_1
C_2	V_2

which means that variable assumes value V_1 when condition C_1 is true or value V_2 when condition C_2 is true, or

Event	Variable
E_1	V_1
E_2	V_2

which means that variable assumes value V_1 upon occurrence of event E_1 or value V_2 upon occurrence of event E_2.

App.2.1 CoRE Information Model

Figure 44 illustrates the CoRE information model.

The following data dictionary entries define the attributes of entities in the information model:

```
Air = mon_Wind_Direction + mon_Wind_Magnitude + mon_Air_Temperature.

Buoy = mon_Buoy_Location.

Light = con_Red_Light.

OmegaGroundUnit = mon_Omega_Error.

Sailor = mon_Emergency_Button.

Vessel = mon_Reset_SOS  + mon_Light_Command
  + mon_Vessel_Request + con_Report.

Water = mon_Water_Temperature.
```

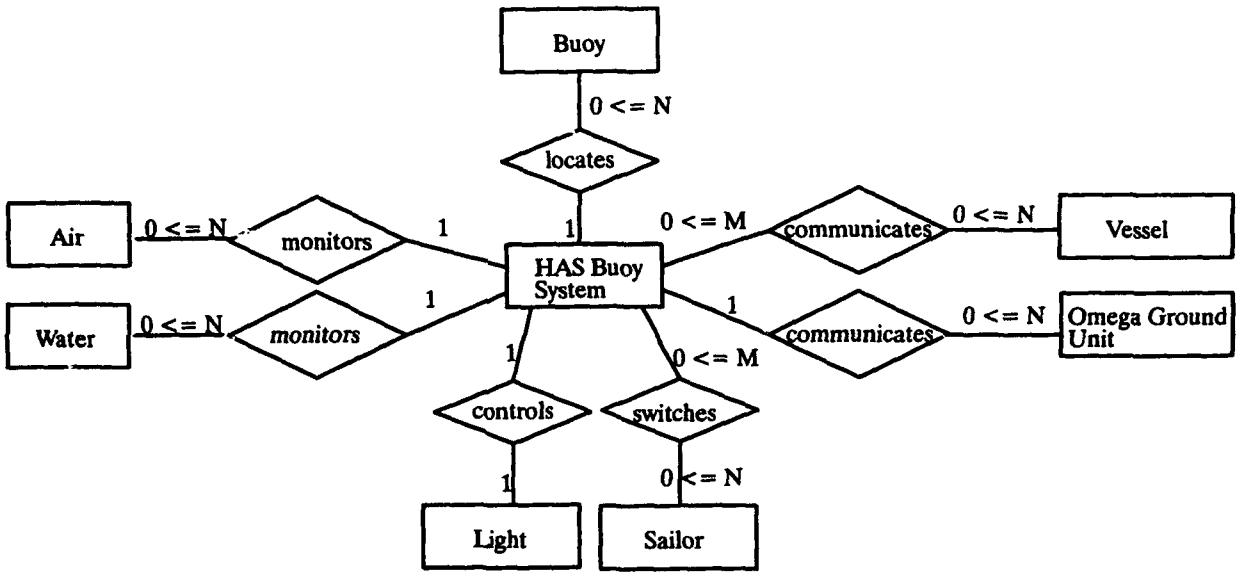


Figure 44. CoRE Information Model

App.2.2 Context Diagram

Figure 45 illustrates the CoRE context diagram.

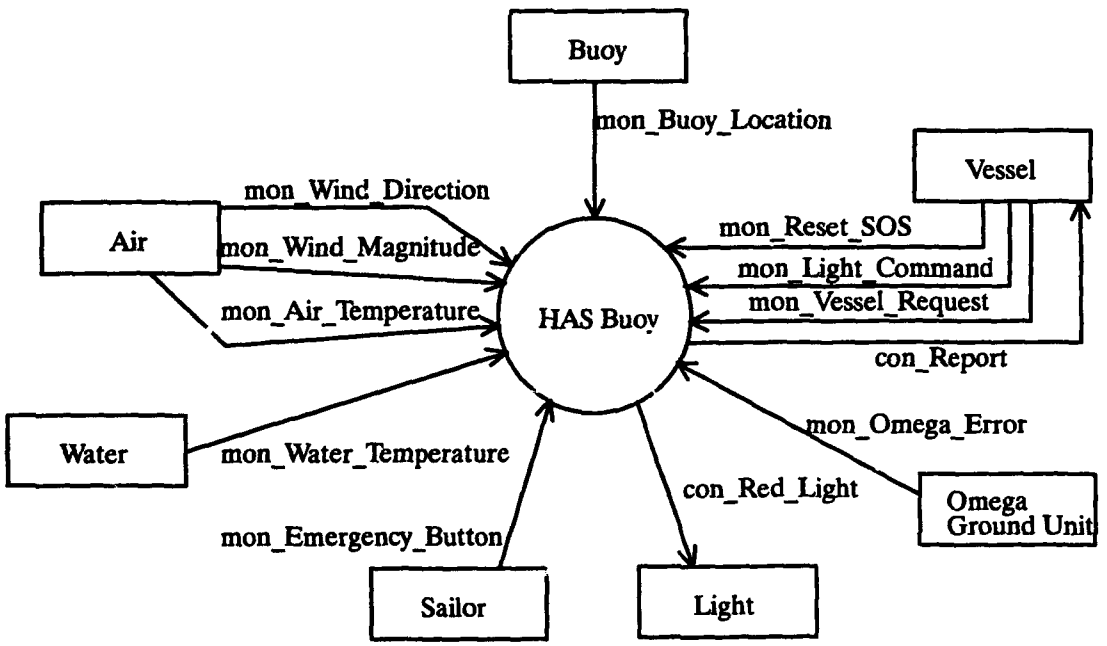


Figure 45. Context Diagram

App.2.3 Dependency Graph

Figure 46 illustrates the CoRE dependency graph.

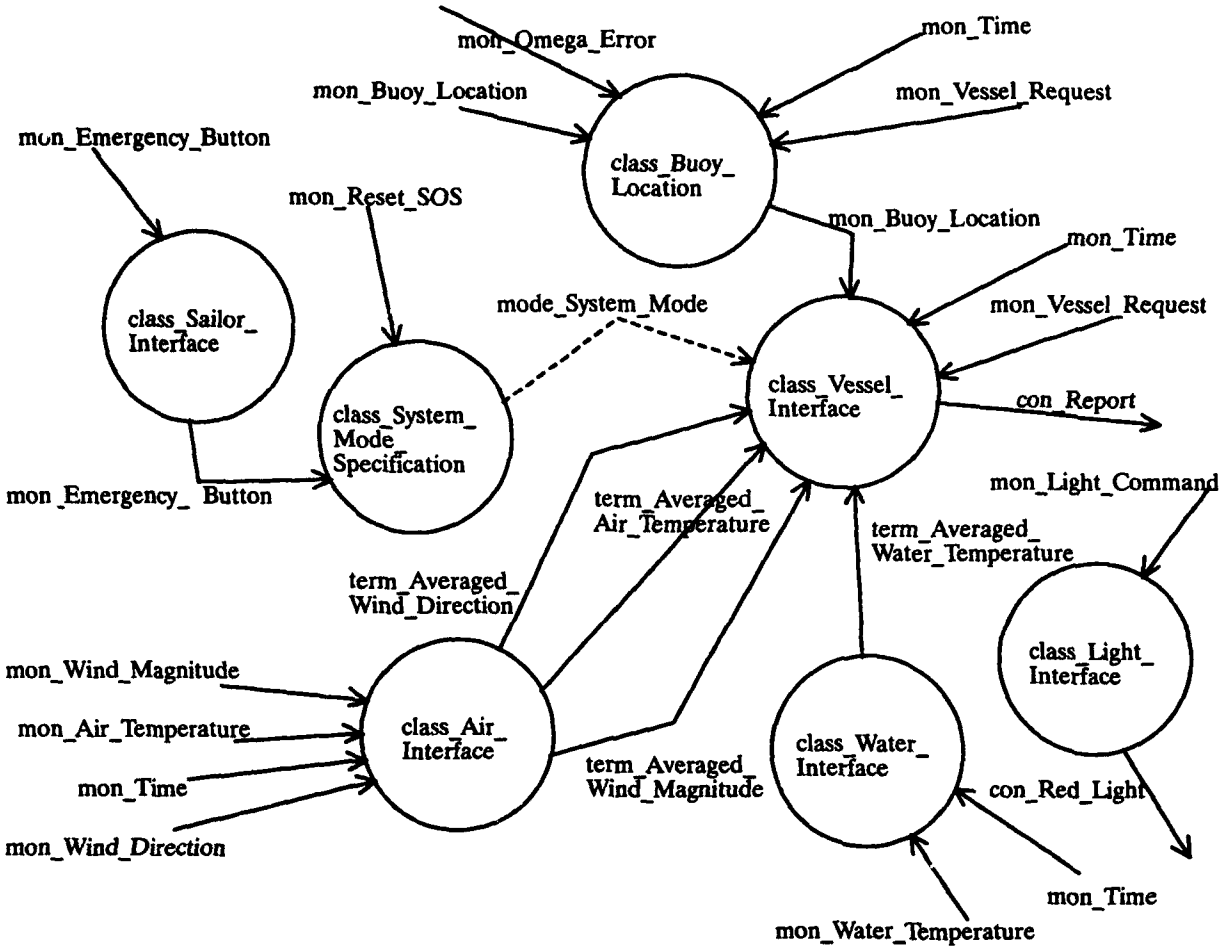


Figure 46. Dependency Graph

App.2.4 Monitored and Controlled Variable Definitions

This section contains the *teamwork* data dictionary entries defining monitored and controlled variables.

```
con_Report = Report_Type + ASCII_Report.
   ------------------
   PhysicalInterpretation
    while Report_Type=SOS_Report broadcasting data defined by DDE
         SOS_Data
    while Report_Type=Wind_and_Temperature_Report broadcasting data
         defined by DDE Wind_and_Temperature_Data
    while Report_Type=Weather_History_Report broadcasting data defined by
         DDE Weather_History_Data
    while Report_Type=Airplane_Detailed_Report broadcasting data defined
     by DDE Airplane_Detailed_Data
    while Report_Type=Ship_Detailed_Report broadcasting data defined by
         DDE Ship_Detailed_Data
    while Report_Type=None, no broadcasting

con_Red_Light = [ "On" | "Off" ].
   ------------------
```

```
   PhysicalInterpretation if con_Red_Light=On the buoy light is on
          if con_Red_Light=Off the buoy light is off

mon_Air_Temperature = Temperature.
   -------------------
   PhysicalInterpretation The temperature of the air ten feet above the
          surface of the water in degrees centigrade.

mon_Buoy_Location = Location.
   -------------------
   PhysicalInterpretation Location of buoy on the earth.

mon_Emergency_Button = [ "Pressed" | "Released"].
   -------------------
   PhysicalInterpretation
     if EmergencyButton = "Pressed," then the button is pressed,
     if EmergencyButton = "Released," then the button is not pressed.

mon_Light_Command = [ "Red_Light_On" | "Red_Light_Off"].
   --------
   PhysicalInterpretation The Request from a passing ship to turn the
          buoy light on (to find the buoy) or off.

mon_Omega_Error =
   <Lat_Offset>Error_Correction + <Lon_Offset>Error_Correction.
   --------
   PhysicalInterpretation
         The correction needed to more accurately determine location from
     the Omega broadcasts, based upon the Omega ground unit monitoring
     the Omega transmissions.

mon_Reset_SOS = [ "True" | "False" ].
   --------
   PhysicalInterpretation  A passing vehicle requests that the SOS signal
          stop broadcasting.

mon_Time = [ "t0" | "Startup" | t ].
   --------
   PhysicalInterpretation  The elapsed time since system startup.

mon_Vessel_Request = [ "Airplane_Detailed_Report_Request" |
       "Ship_Detailed_Report_Request" | "History_Report_Request"].
   -------------------
   PhysicalInterpretation
     Represents requests for reports from passing vessels.

mon_Water_Temperature = Temperature.
   ---------------------
   Values     -4 <= mon_Water_Temperature <= 100
   PhysicalInterpretation The temperature of the water four feet below the
            surface of the water in degrees centigrade.

mon_Wind_Direction = Direction.
   -------------------
```

```
   PhysicalInterpretation The direction the wind is blowing measured 10
         feet above the surface of the water.

mon_Wind_Magnitude = Magnitude.
   -------------------
   PhysicalInterpretation
    The speed of the wind in nautical miles per hour, measured
    10 feet above the surface of the water.
```

App.2.5 Input and Output Variables

This section contains the **team***work* data dictionary entries defining input and output variables.

```
in_Air_Temperature_Sensor = BYTE
   ------------------
   Hardware    Air temperature sensors
   Values    -128 <= in_Air_Temperature_Sensor <= 127
   DataTransfer   Ports B1 and B2
   DataRepresentation 8-bits, two's-complement integer

in_Button_Indicator = [ "2#1xxxxxxx#" | "2#0xxxxxxx#"]
   --------
   Hardware    Emergency button on buoy
   Values    see above
   DataTransfer   Port E
   DataRepresentation 8-bits

in_Incoming_Radio_Message =
   [ "1" * Red_Light_On *
    | "2" * Red_Light_Off *
    | "3" * History_Report_Request *
    | "4" * Airplane_Detailed_Report_Request *
    | "5" * Ship_Detailed_Report_Request *
    | "6" * Terminate_SOS_Signal *
    | "7"]. * Location_Correction_Request +
      in_Location_Correction_Data, see it's DDE *
   --------
   Hardware    Radio receiver
   Values
     Byte 1 indicates one of None, Red_Light_On, Red_Light_Off,
      History_Report_Request, Airplane_Detailed_Report_Request,
      Ship_Detailed_Report_Request, Terminate_SOS_Signal, or
      Location_Correction_Request.
      in_Location_Correction_Data(Bytes 2 & 3): Contains
      Location_Correction_Data when Byte 1 indicates
      Location_Correction_Request, otherwise, Bytes 2 and 3
      are unused.
   DataTransfer   Port F
   DataRepresentation
     3 bytes: Byte 1: 16#01# = Red_Light_On,
         16#02# = Red_Light_Off,
         16#03# = History_Report_Request,
         16#04# = Airplane_Detailed_Report_Request,
```

```
          16#05# = Ship_Detailed_Report_Request,
          16#06# = Terminate_SOS_Signal,
          16#07# = Location_Correction_Request,
          others = None.
        Byte 2: if Byte 1 = Location_Correction_Request then:
            8-bit two's complement integer representing
            latitude Omega error in kilometers
            otherwise unused.
        Byte 3: if Byte 1 = Location_Correction_Request then:
            8-bit two's complement integer representing
            longitude Omega error in kilometers
            otherwise unused.

in_Location_Correction_Data = <u>BYTE + <l>BYTE.
   --------
   Hardware    Radio receiver
   Values    -128 <= in_Location_Correction_Data.u <= 127
         -128 <= in_Location_Correction_Data.l <= 127
   DataTransfer   Port F
   DataRepresentation 2 Bytes: see DDE for in_Incoming_Radio_Message

in_Omega_System_Input = <Latitude>Digital_Angle + <Longitude>Digital_Angle
   --------
   Hardware    Omega navigation system
   Values    see DDE for Digital_Angle
   DataTransfer   Port D
   DataRepresentation see DDE for Digital_Angle

IN_Relation_for_mon_Time =
   mon_Time = in_Time (and in_Time = ~mon_Time)
   -------------
   mon_Time is the system time elapsed since startup.

in_Time =
   --------
   Hardware    system clock
   Values    see DDE for mon_Time
   DataTransfer   supplied by run-time system
   DataRepresentation supplied by run-time system

in_Water_Temperature_Sensor = BYTE
   --------
   Hardware    Water temperature sensors
   Values    -128 <= in_Water_Temperature_Sensor <= 127
   DataTransfer   Ports A1 and A2
   DataRepresentation 8-bits, two's-complement integer

in_Wind_Sensors = <North>Sensor + <South>Sensor +
                  <East>Sensor + <West>Sensor
   ------------------
   Note: in_Wind_Sensors is indexed by the direction corresponding to one
     of the sensors, North | South | East | West. Each sensor
     measures the force of the wind coming from its respective
     direction. Note that any force on a sensor means the opposing
```

```
      sensor should not register any value (not ((N>0 and S>0) or
      (E>0 and W>0))).

   Hardware    Four wind sensors
   Values    0 <= <*>in_Wind_Sensor <= 255
   DataTransfer   Ports C1 (North sensor), C2 (South sensor),
         C3 (East sensor), and C4 (West sensor)
   DataRepresentation Each port has 8-bits, unsigned integer

out_Light_Switch = [ "2#1xxxxxxx#" | "2#0xxxxxxx#"].
   --------
   Hardware    Buoy light
   Values    see above
   DataTransfer   Port H
   DataRepresentation 8-bits

out_Outgoing_Radio_Message = Report_Code + Page_Count + Page_of_Text.
   --------
   Hardware    Radio transmitter
   DataTransfer   Port G
   DataRepresentation record
         Report_Code at 0  range 0..7
         Page_Count at 1 Byte range 0..7
         Page_of_Text at 2 Byte range 0..510 x 8
         end record
```

App.2.6 Event and Term Definitions

This section contains the **team***work* data dictionary entries defining events and terms. These terms and events are referenced by other CoRE artifacts. Note that some of the events are used in the definitions of the inverse of value functions (not REQ, IN, or OUT relations). Also, in some cases, the definitions of events are incomplete — some do not define frequency profile.

```
event_Airplane_Detailed_Report_Request =
   @T(mon_Vessel_Request = "Airplane_Detailed_Report_Request")
   ----------------------
   FrequencyProfile
    MinimumInterval  1.0 second
    ExpectedInterval  30 minutes
    MaximumInterval  N/A

event_Button_Indicator_Reset = @T(in_Button_Indicator = "2#0xxxxxxx#")

event_Button_Indicator_Set = @T(in_Button_Indicator = "2#1xxxxxxx#")

event_Emergency_Button_Pressed = @T(mon_Emergency_Button = "Pressed")

event_Emergency_Button_Released = @F(mon_Emergency_Button = "Pressed")

event_History_Report_Request =
   @T(mon_Vessel_Request = "History_Report_Request")
   ---------------
   FrequencyProfile
    MinimumInterval  1.0 second
```

```
    ExpectedInterval  30 minutes
    MaximumInterval  N/A

event_Incoming_Radio_Message_1 = @T(in_Incoming_Radio_Message = "1")

event_Incoming_Radio_Message_2 = @T(in_Incoming_Radio_Message = "2")

event_Incoming_Radio_Message_3 = @T(in_Incoming_Radio_Message = "3")

event_Incoming_Radio_Message_4 = @T(in_Incoming_Radio_Message = "4")

event_Incoming_Radio_Message_5 = @T(in_Incoming_Radio_Message = "5")

event_Incoming_Radio_Message_6 = @T(in_Incoming_Radio_Message = "6")

event_Incoming_Radio_Message_7 = @T(in_Incoming_Radio_Message = "7")

event_Omega_Update = @T(mon_Omega_Error(t) /= mon_Omega_Error(t - 1))

event_Outgoing_Radio_Message =
   @F(out_Outgoing_Radio_Message.Report_Code = "2#0xxxxxxx#")

event_Periodic_30_Second = @T([mon_Time MOD 30 seconds] = 0)

event_Periodic_60_Second = @T([mon_Time MOD 60 seconds] = 0)
   --------------------
   Period
    MinimumInterval  57.5 seconds
    ExpectedInterval 60.0 seconds
    MaximumInterval  62.5 seconds

event_Red_Light_Off = @T(mon_Light_Command = "Red_Light_Off")
   -------------
   FrequencyProfile
    MinimumInterval  10 seconds
    ExpectedInterval 30 minutes
    MaximumInterval  N/A

event_Red_Light_On = @T(mon_Light_Command = "Red_Light_On")
   -----------------
   FrequencyProfile
    MinimumInterval  10 seconds
    ExpectedInterval 30 minutes
    MaximumInterval  N/A

event_Report_Available = @F(~con_Report.Report_Type = "None")

event_Reset_SOS = @T(mon_Reset_SOS = "True")

event_Ship_Detailed_Report_Request =
   @T(mon_Vessel_Request = "Ship_Detailed_Report_Request")
   -------------------
   FrequencyProfile
    MinimumInterval  1.0 second
    ExpectedInterval 30 minutes
    MaximumInterval  N/A
```

```
term_Airplane_Detailed_Report =
   mon_Buoy_Location
   + term_Averaged_Air_Temperature
   + term_Averaged_Water_Temperature
   + term_Averaged_Wind_Direction
   + term_Averaged_Wind_Magnitude.

term_Averaged_Air_Temperature =
   ROUND [(SUM i: 0 <= i <= 5 : mon_Air_Temperature (t - 10 x i)) / 6]
   --------------
   Since there are two air temperature sensors,
    term_Averaged_Air_Temperature is in fact the averaged air
    temperatures from the two sensors.

term_Averaged_Water_Temperature =
   ROUND [(SUM i: 0 <= i <= 5 : mon_Water_Temperature (t - 10 x i)) / 6]
   --------------
   Since there are two water temperature sensors,
    term_Averaged_Water_Temperature must in fact average the averaged
    water temperatures from the two sensors.

term_Averaged_Wind_Direction =
   Angle_Of (VECTOR_SUM {VECTOR (mon_Wind_Direction (t),
           mon_Wind_Direction (t - 30))} / 2)

term_Averaged_Wind_Magnitude =
   ROUND ([mon_Wind_Magnitude(t - 30) + mon_Wind_Magnitude(t)] / 2)

term_Ship_Detailed_Report (control flow) =
   mon_Buoy_Location
   + term_Averaged_Air_Temperature
   + term_Averaged_Water_Temperature
   + term_Averaged_Wind_Direction
   + term_Averaged_Wind_Magnitude.

term_SOS_Report = mon_Buoy_Location.

term_Weather_History_Report =
   * The set of term_Wind_and_Temperature_Report(i), where i = t-136_800,
    t-136_740, ..., t (i.e., step by 60 seconds). That is, the
    term_Wind_and_Temperature_Report at every 60 second interval over the
    last 48 hours. *

term_Wind_and_Temperature_Report =
   term_Averaged_Air_Temperature
   + term_Averaged_Water_Temperature
   + term_Averaged_Wind_Direction
   + term_Averaged_Wind_Magnitude.

term_Wind_Vector =
   * This term is used in the description of how mon_Wind_Magnitude and
    mon_Wind_Direction relate to in_Wind_Sensors. *
   <X>Magnitude + <Y>Magnitude.
   --------
```

```
Using conventions, let North be the Y-axis, and East be the X-axis.
Then by the definition of mon_Wind_Direction, it is the angle in degrees
 measured clockwise from North.

Therefore, the equivalent vector, (x,y) is defined by:
 x = mon_Wind_Magnitude x SIN(360 - mon_Wind_Direction),
 y = mon_Wind_Magnitude x COS(360 - mon_Wind_Direction).

Which can be simplified to,
 x = - mon_Wind_Magnitude x SIN(mon_Wind_Direction),
 y = mon_Wind_Magnitude x COS(mon_Wind_Direction).
-------------------
Notes:

~mon_Wind_Magnitude in knots =
   SQRT(SQUARE(in_Wind_Sensor.South + in_Wind_Sensor.North) +
    SQUARE(in_Wind_Sensor.West + in_Wind_Sensor.East ))

~mon_Wind_Direction in degrees (where 0=North,  90=East,
         180=South, and 270=West) =
 if (in_Wind_Sensor.West > 0) AND (in_Wind_Sensor.South > 0) then
  = ROUND(INVTAN(in_Wind_Sensor.West / in_Wind_Sensor.South) + 180)
 elsif (in_Wind_Sensor.West > 0) AND (in_Wind_Sensor.North > 0) then
  = ROUND(INVTAN(in_Wind_Sensor.North / in_Wind_Sensor.West) + 270)
 elsif (in_Wind_Sensor.East > 0) AND (in_Wind_Sensor.North > 0) then
  = ROUND(INVTAN(in_Wind_Sensor.East / in_Wind_Sensor.North))
 elsif (in_Wind_Sensor.East > 0) AND (in_Wind_Sensor.South > 0) then
  = ROUND(INVTAN(in_Wind_Sensor.South / in_Wind_Sensor.East) + 90)
 elsif (in_Wind_Sensor.West <= 0) AND (in_Wind_Sensor.East <= 0) AND
  (in_Wind_Sensor.South > 0) then
  = 180
 elsif (in_Wind_Sensor.West <= 0) AND (in_Wind_Sensor.East <= 0) AND
  (in_Wind_Sensor.South <= 0) then
  = 0
```

App.2.7 CLASS_SYSTEM_MODE_SPECIFICATION

The class_System_Mode_Specification requirements class encapsulates the mode machine for the HAS Buoy and the monitored variable mon_Reset_SOS.

App.2.7.1 *Mode Machines*

Figure 47 illustrates the HAS Buoy mode machine in the form of a state-transition diagram.

```
mode_System_Mode = [ "mode_SOS" | "mode_Normal" ].
   --------------------
   Initial Value   "mode_Normal"
   PhysicalInterpretation
     The current state of the system:
        "mode_SOS" = currently transmitting SOS signals,
        "mode_Normal" = all other times.
```

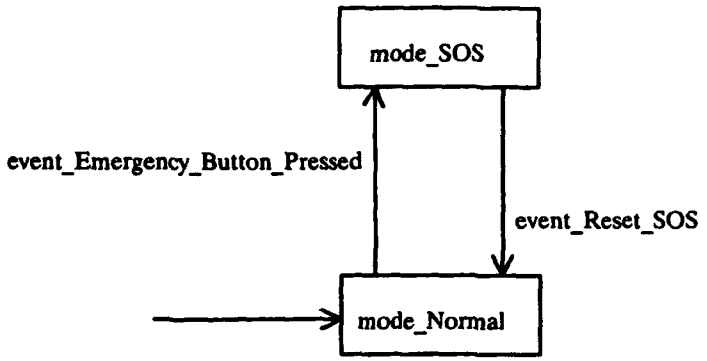


Figure 47. Mode Machine for mode_System_Mode

App.2.7.2 IN and OUT Relations

Figure 48 illustrates the IN relation for mon_Reset_SOS. Figure 49 illustrates the inverse of the IN value function for mon_Reset_SOS.

Event	in_Incoming_Radio_Message
event_Reset_SOS	"6"

Figure 48. IN Relation for mon_Reset_SOS

Event	~mon_Reset_SOS
event_Incoming_Radio_Message_6	"True"

Figure 49. IN' for mon_Reset_SOS

```
timing_Radio_Receiver =
   Device  "Active"
      Events: event_Airplane_Detailed_Report_Request,
        event_Ship_Detailed_Report_Request,
        event_History_Report_Request,
        event_Red_Light_On,
        event_Red_Light_Off,
        event_Reset_SOS,
        event_Omega_Update
   Tolerance N/A
   Delay  6.0 seconds
```

App.2.8 CLASS_AIR_INTERFACE

The class_Air_Interface requirements class encapsulates the monitored variables mon_Air_Temperature, mon_Wind_Magnitude, and mon_Wind_Direction.

App.2.8.1 IN and OUT Relations

Figure 50 illustrates the IN relation for mon_Air_Temperature. Figure 51 illustrates the inverse of the IN value function for mon_Air_Temperature.

Condition	in_Air_Temperature_Sensor
True	TRUNCATE (256 x (mon_Air_Temperature + 100) / 200) − 128

Figure 50. IN Relation for mon_Air_Temperature

Condition	~mon_Air_Temperature
True	(200 x (in_Air_Temperature_Sensor + 128) / 256) − 100

Figure 51. IN' for mon_Air_Temperature

```
timing_Air_Temperature_Sensor =
   Device  "Passive"
   Tolerance
   Delay  1 second
```

Figure 52 illustrates the IN relation for mon_Wind. Figure 53 illustrates the inverse of the IN value function for mon_Wind.

```
timing_Wind_Sensors =
   Device  "Passive"
   Tolerance
   Delay  1 second
```

App.2.8.2 NAT Relations

```
(d mon_Air_Temperature / dt) < MAX_RATE_AIR_TEMPERATURE_CHANGE

(d mon_Wind_Direction / dt) < MAX_RATE_WIND_DIRECTION_CHANGE

(d mon_Wind_Magnitude / dt) < MAX_RATE_WIND_MAGNITUDE_CHANGE
```

App.2.9 CLASS_WATER_INTERFACE

class_Water_Interface encapsulates the monitored variable mon_Water_Temperature.

<Y>term_Wind_Vector >=0	<X>term_Wind_Vector >=0	<North>in_Wind_ Sensors	<South>in_Wind_ Sensors	<East>in_Wind_ Sensors	<West>in_Wind_ Sensors
"TRUE"	"TRUE"	term_Wind_Vector.Y	0	term_Wind_Vector.X	0
"FALSE"	"TRUE"	0	– term_Wind_Vector.Y	term_Wind_Vector.X	0
"TRUE"	"FALSE"	term_Wind_Vector.Y	0	0	– term_Wind_Vector.X
"FALSE"	"FALSE"	0	– term_Wind_Vector.Y	0	– term_Wind_Vector.X

Figure 52. IN Relation for mon_Wind

Condition	~term_Wind_Vector
NOT (((<North>in_Wind_Sensors > 0) AND (<South>in_Wind_Sensors > 0)) OR ((<East>in_Wind_Sensors > 0) AND (<West>in_Wind_Sensors > 0)))	(X <= ((<South>in_Wind_Sensors + <North>in_Wind_Sensors) ** 2), Y <= ((<East>in_Wind_Sensors + <West>in_Wind_Sensors) ** 2))
((<North>in_Wind_Sensors > 0) AND (<South>in_Wind_Sensors > 0)) OR ((<East>in_Wind_Sensors > 0) AND (<West>in_Wind_Sensors > 0))	* Device error *

Figure 53. IN' for mon_Wind

App.2.9.1 IN and OUT Relations

Figure 54 illustrates the IN relation for mon_Water_Temperature. Figure 55 illustrates the inverse of the IN value function for mon_Water_Temperature.

Condition	in_Water_Temperature_Sensor
True	TRUNCATE(255 x (mon_Water_Temperature + 4) / 104)

Figure 54. IN Relation for mon_Water_Temperature

Condition	~ mon_Water_Temperature
True	(104 x in_Water_Temperature_Sensor / 255) − 4

Figure 55. IN' for mon_Water_Temperature

```
timing_Water_Temperature_Sensor =
   Device  "Passive"
   Tolerance
   Delay  1 second
```

App.2.9.2 NAT Relations

```
-4 <= mon_Water_Temperature <= 100 (degrees Celsius)

(d mon_Water_Temperature / dt) < MAX_RATE_WATER_TEMPERATURE_CHANGE
```

App.2.10 CLASS_BUOY_LOCATION

The class_Buoy_Location requirements class encapsulates the monitored variables mon_Buoy_Location and mon_Omega_Error.

App.2.10.1 IN and OUT Relations

Figure 56 illustrates the IN relation for mon_Buoy_Location. Figure 57 illustrates the inverse of the IN value function for mon_Buoy_Location.

Event	in_Omega_System_Input
event_Periodic_30_Second	|(mon_Buoy_Location + mon_Omega_Error) − in_Omega_System_Input| < 0.4 km

Figure 56. IN Relation for mon_Buoy_Location

Event	~mon_Buoy_Location
event_Periodic_30_Second	Latitude <= (Degrees <=MAX(Latitude>in_Omega_System_Input.Bytes_1&2,359), Minutes <= MAX(<Latitude>in_Omega_System_Input.Byte_3, 59), Seconds <= MAX(<Latitude>in_Omega_System_Input.Byte_4, 59) + MAX(<Latitude>in_Omega_System_Input.Byte_5, 99) / 100), Longitude <= Degrees <= MAX<Longitude>in_Omega_System_Input.Byte_1&2,359 Minutes <= MAX(<Longitude>in_Omega_System_Input.Byte_3, 59), Seconds <= MAX(<Longitude>in_Omega_System_Input.Byte_4, 59) + MAX(<Longitude>in_Omega_System_Input.Byte_5, 99) / 100)

Figure 57. IN' for mon_Buoy_Location

```
timing_Omega_System =
   Device  "Periodic" @ 30 seconds
   Tolerance
   Delay  10 seconds
```

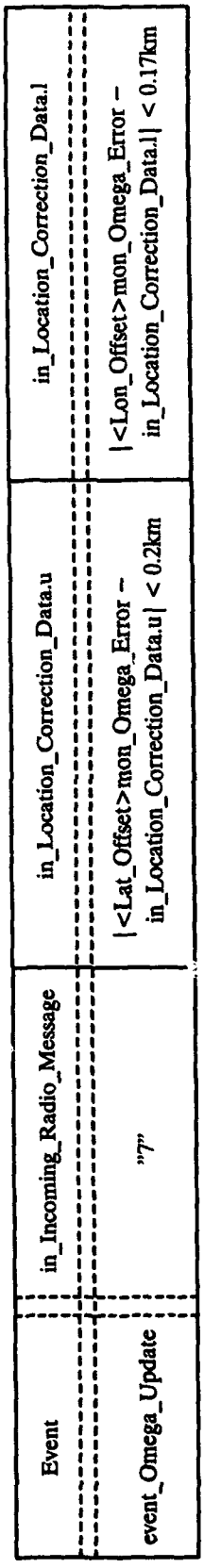
Figure 58 illustrates the IN relation for mon_Omega_Error. Figure 59 illustrates the inverse of the IN value function for mon_Omega_Error.

App.2.10.2 NAT Relations

```
(d mon_Buoy_Location / dt) < MAX_CHANGE_LOCATION
```

App.2.11 CLASS_VESSEL_INTERFACE

The class_Vessel_Interface requirements class encapsulates monitored variable mon_Vessel_Request and controlled variable con_Report.



Event	in_Incoming_Radio_Message	in_Location_Correction_Data.u	in_Location_Correction_Data.l
event_Omega_Update	"7"	|<Lat_Offset>mon_Omega_Error – in_Location_Correction_Data.u| < 0.2km	|<Lon_Offset>mon_Omega_Error – in_Location_Correction_Data.l| < 0.17km

Figure 58. IN Relation for mon_Omega_Error

Event	~mon_Omega_Error
event_Incoming_Radio_Message_7	Lat_Offset <= in_Location_Correction_Data.u, Lon_Offset <= in_Location_Correction_Data.l

Figure 59. IN' for mon_Omega_Error

App.2.11.1 REQ Relations

Figure 60 illustrates the REQ relation for con_Report. The NAT_Relation_for_con_Report_Timing makes the events in this table effectively synchronous.

```
behavior_of_con_Report =
   ------------------------------------
   --  Scheduling Constraints  --
   ------------------------------------
   ControlledVariable  con_Report
   InitialValue     "None"
   ModeClass     Mode_Class_for_mode_System_Mode
   SustainingConditions  N/A
   ValueFunction    see REQ_Relation_for_con_Report
   Tolerance     see individual monitored variables that are
         used to build the different kinds of reports
   NATConstraints    N/A
   InitiationDelay    7.5 seconds
   ------------------------------------
   -- Periodic Scheduling Constraints --
   ------------------------------------
   Events      event_Periodic_60_Second
   InitiationTermination initiated upon expiration of InitiationDelay,
         never terminates
   CompletionDeadline  for event_Periodic_60_Second, 5.0 seconds
   ---------------------------------
   -- Demand Scheduling Constraints --
   ---------------------------------
   Events      event_Ship_Detailed_Report_Request,
         event_Airplane_Detailed_Report_Request,
         event_History_Report_Request
   CompletionDeadline for event_Ship_Detailed_Report_Request, 5.0 minutes
        for event_Airplane_Detailed_Report_Request, 2.0 min.
        for event_History_Report_Request, 6.0 minutes
```

Mode	Event	Event	Event	Event	Event
"mode_SOS"	event_Periodic_ 60_Second	X	event_Airplane_Detailed_ Report_Request	event_Ship_Detailed_ Report_Request	event_History_ Report_Request
"mode_Normal"	X	event_Periodic_60_Second	event_Airplane_Detailed_ Report_Request	event_Ship_Detailed_ Report_Request	event_History_ Report_Request
con_Report.Report_ Type	"SOS_Report"	"Wind_and_Temperature_Report"	"Airplane_Detailed_Report"	"Ship_Detailed_Report"	"Weather_History_Report"
con_Report.ASCII_ Report	ASCII (term_ SOS_Report)	ASCII (term_Wind_ and_Temperature_Report)	ASCII (term_Airplane_ Detailed_Report)	ASCII (term_Ship_ Detailed_Report)	ASCII (term_Weather_ History_Report)

Figure 60. REQ Relation for con_Report

App.2.11.2 IN and OUT Relations

Figure 61 illustrates the IN relation for mon_Vessel_Request. Figure 62 illustrates the inverse of the IN value function for mon_Vessel_Request.

Event	in_Incoming_Radio_Message
event_History_Report_Request	"3"
event_Airplane_Detailed_Report_Request	"4"
event_Ship_Detailed_Report_Request	"5"

Figure 61. IN Relation for mon_Vessel_Request

Event	~mon_Vessel_Request
event_Incoming_Radio_Message_3	"History_Report_Request"
event_Incoming_Radio_Message_4	"Airplane_Detailed_Report_Request"
event_Incoming_Radio_Message_5	"Ship_Detailed_Report_Request"

Figure 62. IN' for mon_Vessel_Request

Section App.2.7.2 contains the specification of timing information related to the radio receiver.

Figure 63 illustrates the OUT relation for con_Report. Figure 64 illustrates the inverse of the OUT value function for con_Report. Each ~con_Report maps to con_Report.ASCII_Report.Number_of_Pages out_Outgoing_Radio_Messages. Iterator identifies the number within that range.

```
timing_Radio_Transmitter =
   Device  "Active"
       Events: event_Outgoing_Radio_Message
   Tolerance
   Delay  10 seconds
```

Event	con_Report.Report_Type	con_Report.ASCII_Report
event_Outgoing_Radio_Message when out_Outgoing_Radio_Message.Report_Code = "2#10000001#"	"SOS_Report"	out_Outgoing_Radio_Message.Page_of_Text (1..out_Outgoing_Radio_Message.Page_Count)
event_Outgoing_Radio_Message when out_Outgoing_Radio_Message.Report_Code = "2#10000010#"	"Wind_and_Temperature_Report"	out_Outgoing_Radio_Message.Page_of_Text (1..out_Outgoing_Radio_Message.Page_Count)
event_Outgoing_Radio_Message when out_Outgoing_Radio_Message.Report_Code = "2#10000011#"	"Airplane_Detailed_Report"	out_Outgoing_Radio_Message.Page_of_Text (1..out_Outgoing_Radio_Message.Page_Count)
event_Outgoing_Radio_Message when out_Outgoing_Radio_Message.Report_Code = "2#10000100#"	"Ship_Detailed_Report"	out_Outgoing_Radio_Message.Page_of_Text (1..out_Outgoing_Radio_Message.Page_Count)
event_Outgoing_Radio_Message when out_Outgoing_Radio_Message.Report_Code = "2#10000101#"	"Weather_History_Report"	out_Outgoing_Radio_Message.Page_of_Text (1..out_Outgoing_Radio_Message.Page_Count)

Figure 63. OUT Relation for con_Report

Event	out_Outgoing_Radio Message.Report_Code	out_Outgoing_Radio_Message. Page_Count	out_Outgoing_Radio_Message. Page_of_Text
event_Report_Available when ~con_Report.Report_Type = "SOS_Report"	"2#10000001#"	Bits 0–3 <= ~con_Report.ASCII_Report. Number_of_Pages Bits 4–7 <= Iterator (range 1 .. 16)	~con_Report.ASCII_Report. Page_of_Text(Iterator)
event_Report_Available when ~con_Report.Report_Type = "Wind_and_Temperature_Report"	"2#10000010#"	Bits 0–3 <= ~con_Report.ASCII_Report. Number_of_Pages Bits 4–7 <= Iterator (range 1 .. 16)	~con_Report.ASCII_Report. Page_of_Text(Iterator)
event_Report_Available when ~con_Report.Report_Type = "Airplane_Detailed_Report"	"2#10000011#"	Bits 0–3 <= ~con_Report.ASCII_Report. Number_of_Pages Bits 4–7 <= Iterator (range 1 .. 16)	~con_Report.ASCII_Report. Page_of_Text(Iterator)
event_Report_Available when ~con_Report.Report_Type = "Ship_Detailed_Report"	"2#10000100#"	Bits 0–3 <= ~con_Report.ASCII_Report. Number_of_Pages Bits 4–7 <= Iterator (range 1 .. 16)	~con_Report.ASCII_Report. Page_of_Text(Iterator)
event_Report_Available when ~con_Report.Report_Type = "Weather_History_Report"	"2#10000101#"	Bits 0–3 <= ~con_Report.ASCII_Report. Number_of_Pages Bits 4–7 <= Iterator (range 1 .. 16)	~con_Report.ASCII_Report. Page_of_Text(Iterator)

Figure 64. OUT' for con_Report

App.2.11.3 NAT Relations

Figure 65 illustrates a NAT relation for con_Report. This NAT relation ensures that the events in REQ_Relation_for_con_Report do not occur at the same time.

Event	mon_Vessel_Request
event_Periodic_60_Second	"None"

Figure 65. NAT Relation for con_Report_Timing

App.2.12 CLASS_LIGHT_INTERFACE

The class_Light_Interface requirements class encapsulates controlled variable con_Red_Light.

App.2.12.1 REQ Relations

Figure 66 illustrates the REQ relation for con_Red_Light.

Event	con_Red_Light
event_Red_Light_On	"On"
event_Red_Light_Off	"Off"

Figure 66. REQ Relation for con_Red_Light

```
behavior_of_con_Red_Light =
   ----------------------------------
   -- Demand Scheduling Constraints --
   ----------------------------------
   ControlledVariable  con_Red_Light
   InitialValue    "Off"
   ModeClass    N/A
   SustainingConditions  N/A
   ValueFunction   see REQ_Relation_for_con_Red_Light
   Tolerance    N/A
   NATConstraints   N/A
   InitiationDelay   7.5 seconds
   CompletionDeadline  1.25 seconds
   Events     event_Red_Light_On,
          event_Red_Light_Off
```

App.2.12.2 IN and OUT Relations

Figure 67 illustrates the IN relation for mon_Light_Command. Figure 68 illustrates the inverse of the IN value function for mon_Light_Command.

Section App.2.7.2 contains the specification of timing information related to the radio receiver.

Event	in_Incoming_Radio_Message
event_Red_Light_On	"1"
event_Red_Light_Off	"2"

Figure 67. IN Relation for mon_Light_Command

Event	~mon_Light_Command
event_Incoming_Radio_Message_1	"Red_Light_On"
event_Incoming_Radio_Message_2	"Red_Light_Off"

Figure 68. IN' for mon_Light_Command

Figure 69 illustrates the OUT relation for con_Red_Light. Figure 70 illustrates the inverse of the OUT value function for con_Red_Light.

out_Light_Switch	con_Red_Light
"2#1xxxxxx#"	"On"
"2#0xxxxxx#"	"Off"

Figure 69. OUT Relation for con_Red_Light

~con_Red_Light	out_Light_Switch
"On"	"2#1xxxxxx#"
"Off"	"2#0xxxxxx#"

Figure 70. OUT' for con_Red_Light

```
timing_Light_Switch =
   Device  "Continuous"
   Tolerance N/A
   Delay  500ms
```

App.2.13 CLASS_SAILOR_INTERFACE

The class_Sailor_Interface requirements class encapsulates monitored variable mon_Emergency_Button.

App.2.13.1 IN and OUT Relations

Figure 71 illustrates the IN relation for mon_Emergency_Button. Figure 72 illustrates the inverse of the IN value function for mon_Emergency_Button.

Event	in_Button_Indicator
event_Emergency_Button_Pressed	"2#1xxxxxxx#"
event_Emergency_Button_Released	"2#0xxxxxxx#"

Figure 71. IN Relation for mon_Emergency_Button

Event	~mon_Emergency_Button
event_Button_Indicator_Set	"Pressed"
event_Button_Indicator_Reset	"Released"

Figure 72. IN' for mon_Emergency_Button

```
timing_Emergency_Button =
   Device  "Active"
      Events: event_Emergency_Button_Pressed,
        event_Emergency_Button_Released
   Tolerance N/A
   Delay  100ms
```

App.2.14 Other Data Dictionary Entries

This section contains the remaining data dictionary entries.

```
   Angle = Degrees + Minutes + Seconds.

   Angle_Of = Angle_Of ({X,Y}) = COTAN (Y/X), X /= 0

   ASCII = * ASCII(X) : ASCII_Report :=

       NOTE: ASCII(A + B) = ASCII(A) & ASCII(B), where "&" implies string
          concatenation

       if X is of type TEMPERATURE:
        Four ASCII characters, with leading '-' and zeros if necessary,
         representing the temperature in degrees centigrade specified by
         X;

       if X is of type LOCATION: A total of 27 ASCII characters: 13 for
```

```
        latitude, 13 for longitude, separated by one space;
      Latitude and longitude are each represented by the following
         ASCII characters:
       1-3) Degrees, with leading zeros if necessary
       4) The degrees symbol (superscript o)
         5-6) Minutes, with a leading zero if necessary
          7) The minutes symbol (')
       8-9) Whole seconds, with a leading zero if necessary
       10) Decimal point ('.')
      11-12) Hundredths of seconds, with a leading zero if necessary
       13) Seconds symbol ('"').

     if X is of type DIRECTION:
      Three ASCII characters, with leading zeros if necessary,
       representing the direction in degrees from north (0 = north,
       90 = east, 180 = south, 270 = west) specified by X.

     if X is of type MAGNITUDE:
      Three ASCII characters, with leading zeros if necessary,
       representing the magnitude in knots specified by X.*

ASCII_Report = 1 { Page_of_Text } Number_of_Pages.

Degrees = * 0 .. 359 *.

Digital_Angle = * A digital representation of an angle. *
   --------
   DataRepresentation
      5 Bytes:
         Bytes 1&2: degrees latitude range 0 .. 65_535, unsigned integer
      Byte 3: minutes latitude range 0 .. 255, unsigned integer
      Byte 4: whole seconds latitude range 0 .. 255, unsigned integer
      Byte 5: 1/100th seconds latitude range 0 .. 255, unsigned integer

Direction =
   * 0 .. 359 degrees (0 = north, 90 = east, 180 = south, 270 = west). *

Error_Correction = * A measure of length in kilometers, range -128 .. 127. *

InMode =
   ------------------------
    InMode(S) :BOOLEAN := (mode_System_Mode = S)

Location = <Latitude>Angle + <Longitude>Angle .

Magnitude = * Nautical miles per hour, range 0 .. 250. *

Minutes = * 0 .. 59. *

Number_of_Pages = * The total number of Page_of_Text report pages to be
        transmitted (range 1 .. 16). *

Page_Count =
   Bits 0-3: 4-bits range 1 .. 16 representing total number of pages in
```

```
        message
    Bits 4-7: 4-bits range 1 .. 16 representing number of page being
        transmitted

Page_of_Text = * 510 bytes of ASCII text. *

Report_Code = [ "2#10000001#" | "2#10000010#" | "2#10000011#"
      | "2#10000100#" | "2#10000101#" | "2#0xxxxxxx#"] .

Report_Type =
   [ "SOS_Report"     | "Wind_and_Temperature_Report" |
    | "Airplane_Detailed_Report" | "Ship_Detailed_Report"
    | "Weather_History_Report" | "None" ].

Seconds = * 0.00 .. 59.99 *.

Sensor = * range of values 0..255 *

t = * current time *

Temperature = * -100 .. 100 degrees centigrade. *
```

APP.3 PROCESS STRUCTURE

This section contains the ADARTS process structure that was built from the CoRE specification for the HAS Buoy problem. The notation for representing process architecture diagrams is based upon a mapping to **teamwork** Ada Structure Graph notation described in Kirk and Wild (1992).

This section is divided into the following four subsections:

- Section App.3.1 contains the initial process architecture diagram created by mapping artifacts of the CoRE software requirements specification to ADARTS processes.
- Section App.3.2 contains the process behavior specifications corresponding to the processes on the initial process architecture diagram.
- Section App.3.3 contains the process architecture diagram that resulted from applying the ADARTS process clustering criteria to the processes on the initial process architecture diagram.
- Section App.3.4 contains the process behavior specifications corresponding to the processes on the final process architecture diagram.

App.3.1 Initial Process Architecture Diagram

Figure 73 illustrates the ADARTS initial process architecture diagram derived from the CoRE specification. The process behavior specifications in Section App.3.2 describe how the criteria applied in deriving the set of processes. Note that the arrows between processes on the initial process architecture diagram do not indicate message passing or invocation — they simply indicate data dependencies or data flow between processes. Message communication will be specified after clustering processes.

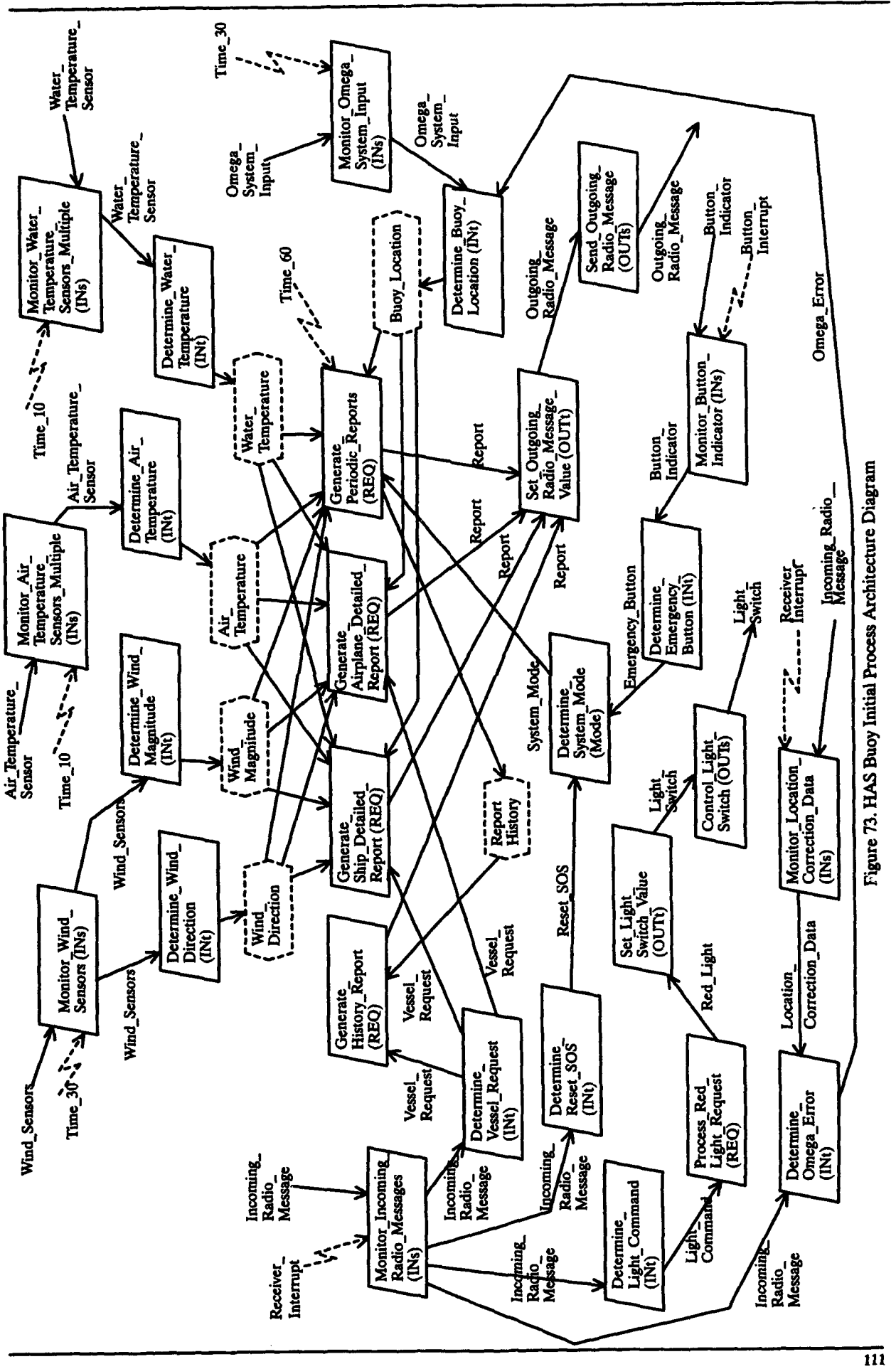


Figure 73. HAS Buoy Initial Process Architecture Diagram

App.3.2 Initial Process Behavior Specifications

This section contains the process behavior specifications associated with the processes on the initial process architecture diagram in Figure 73. Note that **team***work* state-event matrices (SEMs) and process activation tables (PATs) were used to specify the logic of processes in stimulus/response form. Sections App.3.2.1 through App.3.2.27 each contain a process behavior specification corresponding to one of the processes on the initial process architecture diagram.

App.3.2.1 Determine_Wind_Direction

```
Requirements:      IN_Relation_for_mon_Wind,
                   term_Wind_Vector
Criteria:          Determine_Wind_Direction is an INt process
Inputs:            Wind_Sensors message
Outputs:           Wind_Direction data
Frequency:         once per 30 seconds
Execution Time:
Priority:          Medium
Errors Detected:   None
Logic:             See Figure 74
```

Stimulus	Response
received Wind_Sensors	North <-- Wind_Sensors(C1) South <-- Wind_Sensors(C2) East <-- Wind_Sensors(C3) West <-- Wind_Sensors(C4) Wind_Direction <-- if (West > 0) AND (South > 0) then use ROUND(INVTAN(West / South) + 180) elsif (West > 0) AND (North > 0) then use ROUND(INVTAN(North / West) + 270) elsif (East > 0) AND (North > 0) then use ROUND(INVTAN(East / North)) elsif (East > 0) AND (South > 0) then use ROUND(INVTAN(South / East) + 90) elsif (West <= 0) AND (East <= 0) AND (South > 0) then use 180 elsif (West <= 0) AND (East <= 0) AND (South <= 0) then use 0 write Wind_Direction to the Wind_Direction data store

Figure 74. Process Logic for Determine_Wind_Direction

App.3.2.2 Determine_Wind_Magnitude

```
Requirements:      IN_Relation_for_mon_Wind,
                   term_Wind_Vector
Criteria:          Determine_Wind_Magnitude is an INt process
Inputs:            Wind_Sensors message
Outputs:           Wind_Magnitude data
Frequency:         once per 30 seconds
Execution Time:
Priority:          Medium
Errors Detected:   None
Logic:             See Figure 75
```

App.3.2.3 Determine_Air_Temperature

```
Requirements:      IN_Relation_for_mon_Air_Temperature,
Criteria:          Determine_Air_Temperature is an INt process
```

Stimulus	Response
received Wind_Sensors	North <-- Wind_Sensors(C1) South <-- Wind_Sensors(C2) East <-- Wind_Sensors(C3) West <-- Wind_Sensors(C4) Wind_Magnitude <-- if (West > 0) AND (South > 0) use SQRT(SQUARE(South) + SQUARE(West)) elsif (West > 0) AND (North > 0) use SQRT(SQUARE(North) + SQUARE(West)) elsif (East > 0) AND (North > 0) use SQRT(SQUARE(North) + SQUARE(East)) elsif (East > 0) and (South > 0) use SQRT(SQUARE(South) + SQUARE(East)) elsif (West <= 0) AND (East <= 0) AND (South > 0) use South elsif (West <= 0) AND (East <= 0) AND (North > 0) use North elsif (North <= 0) AND (South <= 0) AND (East > 0) use East elsif (North <= 0) AND (South <= 0) AND (West > 0) use West elsif (North <= 0) AND (South <= 0) AND (East <= 0) AND (West <= 0) use 0 write Wind_Magnitude to Wind_Magnitude data store

Figure 75. Process Logic for Determine_Wind_Magnitude

```
Inputs:            Air_Temperature_Sensor message
Outputs:           Air_Temperature data
Frequency:         twice per 10 seconds (1 per air temperature sensor)
Execution Time:
Priority:          Medium
Errors Detected:   None
Logic:             See Figure 76
```

Stimulus	Response
received Air_Temperature_Sensor	Air_Temperature <-- 200 x (Air_Temperature_Sensor + 128) / 256 – 100 write Air_Temperature to the Air_Temperature data store

Figure 76. Process Logic for Determine_Air_Temperature

App.3.2.4 Determine_Water_Temperature

```
Requirements:      IN_Relation_for_mon_Water_Temperature,
Criteria:          Determine_Water_Temperature is an INt process
Inputs:            Water_Temperature_Sensor message
Outputs:           Water_Temperature data
Frequency:         twice per 10 seconds (1 per water temperature sensor)
Execution Time:
Priority:          Medium
Errors Detected:   None
Logic:             See Figure 77
```

Stimulus	Response
received Water_Temperature_Sensor	Water_Temperature <-- (104 x Water_Temperature_Sensor) / 255 – 4 write Water_Temperature to the Water_Temperature data store

Figure 77. Process Logic for Determine_Water_Temperature

App.3.2.5 Determine_Buoy_Location

```
Requirements:       IN_Relation_for_mon_Buoy_Location,
Criteria:           Determine_Buoy_Location is an INt process
Inputs:             Omega_System_Input message,
                    Omega_Error message
Outputs:            Buoy_Location data
Frequency:          once per 30 seconds for Omega_System_Input,
                    see DDE event_Incoming_Radio_Message_7
Execution Time:
Priority:           Medium
Errors Detected:    None
Logic:              See Figure 78
```

Stimulus	Response
received Omega_System_Input	Buoy_Location.Latitude <-- (Degrees <= MAX(<Latitude>Omega_System_Input.Bytes_1&2, 359), Minutes <= MAX(<Latitude>Omega_System_Input.Byte_3, 59), Seconds <= MAX(<Latitude>Omega_System_Input.Byte_4, 59) + MAX(<Latitude>Omega_System_Input.Byte_5, 99) / 100), Buoy_Location.Longitude <-- (Degrees <= MAX(<Longitude>Omega_System_Input.Bytes_1&2, 359), Minutes <= MAX(<Longitude>Omega_System_Input.Byte_3, 59), Seconds <= MAX(<Longitude>Omega_System_Input.Byte_4, 59) + MAX(<Longitude>Omega_System_Input.Byte_5, 99) / 100) Buoy_Location <-- Adjust_for_Error (Buoy_Location, Omega_Error) write Buoy_Location to Buoy_Location data store
received Omega_Error	store Omega_Error locally for future calculations of Buoy_Location

Figure 78. Process Logic for Determine_Buoy_Location

App.3.2.6 Determine_Emergency_Button

```
Requirements:       IN_Relation_for_mon_Emergency_Button,
Criteria:           Determine_Emergency_Button is an INt process
Inputs:             Button_Indicator message
Outputs:            Emergency_Button message
Frequency:          see DDEs for event_Emergency_Button_Pressed and
                    event_Emergency_Button_Released
Execution Time:
Priority:           Medium
Errors Detected:    None
Logic:              See Figure 79
```

Stimulus	Response
received Button_Indicator	if (Button_Indicator = 2#1xxxxxxx#) then Emergency_Button <-- "Pressed" send Emergency_Button to Determine_System_Mode else Emergency_Button <-- "Released"

Figure 79. Process Logic for Determine_Emergency_Button

App.3.2.7 Determine_Vessel_Request

```
Requirements:      IN_Relation_for_mon_Vessel_Request,
Criteria:          Determine_Vessel_Request is an INt process
Inputs:            Incoming_Radio_Message message
Outputs:           Vessel_Request message
Frequency:         see DDEs for event_Incoming_Radio_Message_3,
                   event_Incoming_Radio_Message_4, and
                   event_Incoming_Radio_Message_5
Execution Time:
Priority:          Medium
Errors Detected:   None
Logic:             See Figure 80
```

Stimulus	Response
received Incoming_Radio_Message	case Incoming_Radio_Message is when 3 => Vessel_Request <-- "History_Report_Request" send Vessel_Request to Generate_History_Report when 4 => Vessel_Request <-- "Airplane_Detailed_Report_Request" send Vessel_Request to Generate_Airplane_Detailed_Report when 5 => Vessel_Request <-- "Ship_Detailed_Report_Request" send Vessel_Request to Generate_Ship_Detailed_Report

Figure 80. Process Logic for Determine_Vessel_Request

App.3.2.8 Generate_Periodic_Reports

```
Requirements:      REQ_Relation_for_con_Report,
                   term_SOS_Report,
                   term_Wind_and_Temperature_Report
Criteria:          Generate_Periodic_Reports is a REQ process
Inputs:            System_Mode message,
                   Buoy_Location data,
                   Air_Temperature data,
                   Water_Temperature data,
                   Wind_Magnitude data,
                   Wind_Direction data,
                   Time_60 event
Outputs:           Report message
Frequency:         See DDE for behavior_of_con_Report
Execution Time:
Priority:          High
Errors Detected:   None
Logic:             See Figure 81
```

App.3.2.9 Process_Red_Light_Request

```
Requirements:      REQ_Relation_for_con_Red_Light
Criteria:          Process_Red_Light_Request is a REQ process
Inputs:            Light_Command message
```

Stimulus	Response
when Time_60	if (System_Mode = "mode_SOS") then Report.Report_Type <-- "SOS_Report" SOS_Report <-- read Buoy_Location data store Report.ASCII_Report <-- ASCII(SOS_Report) send Report to Set_Outgoing_Radio_Message_Value elsif (System_Mode = "mode_Normal") then Report.Report_Type <-- "Wind_and_Temperature_Report" read Water_Temperature values from Water_Temperature data store calculate term_Averaged_Water_Temperature read Air_Temperature values from Air_Temperature data store calculate term_Averaged_Air_Temperature read Wind_Direction values from Wind_Direction data store calculate term_Averaged_Wind_Direction read Wind_Magnitude from Wind_Magnitude data store calculate term_Averaged_Wind_Magniutde Wind_and_Temperature_Report <-- (term_Averaged_Water_Temperature, term_Averaged_Air_Temperature, term_Averaged_Wind_Direction, term_Averaged_Wind_Magnitude) Report.ASCII_Report = ASCII(Wind_and_Temperature_Report) send Report to Set_Outgoing_Radio_Message_Value write Report to the Report_History data store
received System_Mode	store current System_Mode locally

Figure 81. Process Logic for Generate_Periodic_Reports

```
Outputs:            Red_Light message
Frequency:          See DDE for behavior_for_con_Red_Light
Execution Time:
Priority:           Medium
Errors Detected:    None
Logic:              See Figure 82
```

Stimulus	Response

received Light_Command	if (mon_Light_Command = "Red_Light_On") then
	Red_Light <-- "On"
	else
	Red_Light <-- "Off"
	send Red_Light to Set_Light_Switch_Value

Figure 82. Process Logic for Process_Red_Light_Request

App.3.2.10 Monitor_Air_Temperature_Sensors_Multiple

```
Requirements:       in_Air_Temperature_Sensor, and
                    IN_Relation_for_mon_Air_Temperature
Criteria:           Monitor_Air_Temperature_Sensors_Multiple is an entity
                    modeling INs process. There is one instance of this
                    process for each of the two air temperature sensors.
Inputs:             Air_Temperature_Sensor data,
                    Time_10 event
Outputs:            Air_Temperature_Sensor message
Frequency:          every 10 seconds
Execution Time:
```

```
Priority:          Medium
Errors Detected:   None
Logic:             See Figure 83
```

Stimulus	Response
when Time_10	Air_Temperature_Sensor <--- Read (RegisterB1 or RegisterB2) send Air_Temperature_Sensor to Determine_Air_Temperature

Figure 83. Process Logic for Monitor_Air_Temperature_Sensors_Multiple

App.3.2.11 Monitor_Wind_Sensors

```
Requirements:      in_Wind_Sensors,
                   IN_Relation_for_mon_Wind
Criteria:          Monitor_Wind_Sensors is an INs process
Inputs:            Wind_Sensors data,
                   Time_30 event
Outputs:           Wind_Sensors message
Frequency:         every 30 seconds
Execution Time:
Priority:          Medium
Errors Detected:   Wind sensors device error
Logic:             See Figure 84
```

Stimulus	Response

when Time_30	Wind_Sensors.North <-- Read(RegiserC1)
	Wind_Sensors.South <--Read(RegisterC2)
	Wind_Sensors.East <-- Read(RegisterC3)
	Wind_Sensors.West <-- Read(RegisterC4)
	if ((Wind_Sensors.North > 0) AND (Wind_Sensors.South > 0)) OR
	((Wind_Sensors.East > 0) AND (Wind_Sensors.West > 0)) then
	there is a device error
	else
	send Wind_Sensors to Determine_Wind_Magnitude
	send Wind_Sensors to Determine_Wind_Direction

Figure 84. Process Logic for Monitor_Wind_Sensors

App.3.2.12 Monitor_Water_Temperature_Sensors_Multiple

```
Requirements:      in_Water_Temperature_Sensor, and
                   IN_Relation_for_mon_Water_Temperature
Criteria:          Monitor_Water_Temperature_Sensors_Multiple is an entity
                   modeling INs process. There is one instance of this
                   process for each of the two water temperature sensors.
Inputs:            Water_Temperature_Sensor data,
                   Time_10 event
Outputs:           Water_Temperature_Sensor message
Frequency:         every 10 seconds
Execution Time:
Priority:          Medium
```

Errors Detected: None
Logic: See Figure 85

Stimulus	Response
when Time_10	Water_Temperature_Sensor <-- Read (RegisterA1 or RegisterA2) send Water_Temperature_Sensor to Determine_Water_Temperature

Figure 85. Process Logic for Monitor_Water_Temperature_Sensors_Multiple

App.3.2.13 Monitor_Location_Correction_Data

Requirements: in_Location_Correction_Data,
IN_Relation_for_mon_Omega_Error
Criteria: Monitor_Location_Correction_Data is an INs process
Inputs: Incoming_Radio_Message data,
Receiver_Interrupt event
Outputs: Location_Correction_Data message
Frequency: See DDE for event_Incoming_Radio_Message_7
Execution Time:
Priority: Medium
Errors Detected: None
Logic: See Figure 86

Stimulus	Response
received Receiver_Interrupt	Read (RegisterF) if (RegisterF.Byte_1 = 16#07#) then Location_Correction_Data.u <-- RegisterF.Byte_2 Location_Correction_Data.l <-- RegisterF.Byte_3 send Location_Correction_Data to Determine_Omega_Error

Figure 86. Process Logic for Monitor_Location_Correction_Data

App.3.2.14 Monitor_Omega_System_Input

Requirements: in_Omega_System_Input,
IN_Relation_for_mon_Buoy_Location
Criteria: Monitor_Omega_System_Input is an INs process
Inputs: Omega_System_Input data,
Time_30 event
Outputs: Omega_System_Input message
Frequency: every 30 seconds
Execution Time:
Priority: Medium
Errors Detected: None
Logic: See Figure 87

App.3.2.15 Monitor_Incoming_Radio_Messages

Requirements: in_Incoming_Radio_Message,
IN_Relation_for_mon_Reset_SOS,

Stimulus	Response
when Time_30	Omega_System_Input <-- read(RegisterD) where (Latitude.Degrees <-- RegisterD.Bytes_1&2 Latitude.Minutes <-- RegisterD.Byte_3 Latitude.Seconds <-- RegisterD.Byte_4 Latitude.Hundredths <-- RegisterD.Byte_5 Longitude.Degrees <--RegisterD.Bytes_6&7 Longitude.Minutes <-- RegisterD.Byte_8 Longitude.Seconds <-- RegisterD.Byte_9 Longitude.Hundredths <-- RegisterD.Byte_10) send Omega_System_Input to Determine_Buoy_Location

Figure 87. Process Logic for Monitor_Omega_System_Input

```
                IN_Relation_for_mon_Omega_Error,
                IN_Relation_for_mon_Vessel_Request,
                IN_Relation_for_mon_Light_Command
Criteria:       Monitor_Incoming_Radio_Messages is an INs process
Inputs:         Incoming_Radio_Message data,
                Receiver_Interrupt event
Outputs:        Incoming_Radio_Message message
Frequency:      see DDEs for event_Incoming_Radio_Message_1,
                event_Incoming_Radio_Message_2,
                event_Incoming_Radio_Message_3,
                event_Incoming_Radio_Message_4,
                event_Incoming_Radio_Message_5,
                and event_Incoming_Radio_Message_6
Execution Time:
Priority:       Medium
Errors Detected: None
Logic:          See Figure 88
```

Stimulus	Response
received Receiver_Interrupt	Read (RegisterF) case RegisterF.Byte_1 is when 16#01# => Incoming_Radio_Message.Byte_1 <-- "Red_Light_On" send Incoming_Radio_Message to Determine_Light_Command when 16#02# => Incoming_Radio_Message.Byte_1 <-- "Red_Light_Off" send Incoming_Radio_Message to Determine_Light_Command when 16#03 => Incoming_Radio_Message.Byte_1 <-- "History_Report_Request" send Incoming_Radio_Message to Determine_Vessel_Request when 16#04# => Incoming_Radio_Message.Byte_1 <-- "Airplane_Detailed_Report_Request" send Incoming_Radio_Message to Determine_Vessel_Request when 16#05# => Incoming_Radio_Message.Byte_1 <-- "Ship_Detailed_Report_Request" send Incoming_Radio_Message to Determine_Vessel_Request when 16#06 => Incoming_Radio_Message.Byte_1 <-- "Terminate_SOS_Signal" send Incoming_Radio_Message to Determine_Reset_SOS when 16#07# => null --handled by Monitor_Location_Correction_Data

Figure 88. Process Logic for Monitor_Incoming_Radio_Messages

App.3.2.16 Monitor_Button_Indicator

```
Requirements:      in_Button_Indicator,
                   IN_Relation_for_mon_Emergency_Button
Criteria:          Monitor_Button_Indicator is an INs process
Inputs:            Button_Indicator data,
                   Button_Interrupt event
Outputs:           Button_Indicator message
Frequency:         see DDEs for event_Emergency_Button_Pressed and
                   event_Emergency_Button_Released
Execution Time:
Priority:          Medium
Errors Detected:   None
Logic:             See Figure 89
```

Stimulus	Response
received Button_Interrupt	read Button_Indicator from RegisterE send Button_Indicator to Determine_Emergency_Button

Figure 89. Process Logic for Monitor_Button_Indicator

App.3.2.17 Set_Outgoing_Radio_Message_Value

```
Requirements:      OUT_Relation_for_con_Report,
Criteria:          Set_Outgoing_Radio_Message_Value is an OUTt process
Inputs:            Report message
Outputs:           Outgoing_Radio_Message message
Frequency:         See DDE for behavior_for_con_Report
Execution Time:
Priority:          High
Errors Detected:   None
Logic:             See Figure 90 (Note: Prioritization of reports must
                   be enforced.)
```

App.3.2.18 Set_Light_Switch_Value

```
Requirements:      OUT_Relation_for_con_Red_Light
Criteria:          Set_Light_Switch_Value is an OUTt process
Inputs:            Red_Light message
Outputs:           Light_Switch message
Frequency:         See DDE for behavior_of_con_Red_Light
Execution Time:
Priority:          Medium
Errors Detected:   None
Logic:             See Figure 91
```

App.3.2.19 Determine_System_Mode

```
Requirements:      Mode_Machine_for_System_Mode
Criteria:          Determine_System_Mode is a mode process
```

Stimulus	Response
received Report	case Report.Report_Type is when "SOS_Report" => Page_Count <-- Length (Report.ASCII_Report) / 510 Outgoing_Radio_Message.Report_Code <-- 2#10000001# Outgoing_Radio_Message.Page_Count.Bits_0-3 <-- Page_Count for Iterator in 1 .. Page_Count loop Outgoing_Radio_Message.Page_Count.Bits4-7 <-- Iterator Outgoing_Radio_Message.Bytes_3-512 <-- Report.ASCII_Report(((Page_Number-1) * 510) + 1 .. (Page_Number * 510)) send Outgoing_Radio_Message to Send_Outgoing_Radio_Message end for loop when "Wind_and_Temperature_Report" => -- do same as for "SOS_Report" except assign 2#10000010# to --- Outgoing_Radio_Message.Report_Code when "Airplane_Detailed_Report" => -- do same as for "SOS_Report" except assign 2#10000011# to -- Outgoing_Radio_Message.Report_Code when "Ship_Detailed_Report" => -- do same as for "SOS_Report" except assign 2#10000100# to -- Outgoing_Radio_Message.Report_Code when "Weather_History_Report" => -- do same as for "SOS_Report" except assign 2#10000101# to -- Outgoing_Radio_Message.Report_Code

Figure 90. Process Logic for Set_Outgoing_Radio_Message_Value

Stimulus	Response
received Red_Light	if (RedLight = "On") then Light_Switch <-- 2#1xxxxxxx# elsif (RedLight = "Off") then Light_Switch <-- 2#0xxxxxxx# send Light_Switch to Control_Light_Switch

Figure 91. Process Logic for Set_Light_Switch_Value

```
Inputs:             Reset_SOS message,
                    Emergency_Button message
Outputs:            System_Mode message
Frequency:          See DDEs for event_Incoming_Radio_Message_6 and
                    event_Emergency_Button_Pressed
Execution Time:
Priority:           Medium
Errors Detected:    None
Logic:              See Figure 92
```

App.3.2.20 Generate_History_Report

```
Requirements:       REQ_Relation_for_con_Report,
                    term_Weather_History_Report
Criteria:           Generate_History_Report is a REQ process
Inputs:             Vessel_Request message,
                    Report_History data
Outputs:            Report message
```

Stimulus	Response
received (Emergency_Button = "Pressed")	if (System_Mode = "mode_Normal") then System_Mode <-- "mode_SOS" send System_Mode to Generate_Periodic_Reports
received (Reset_SOS = "True")	if (System_Mode = "mode_SOS") then System_Mode <-- "mode_Normal" send System_Mode to Generate_Periodic_Reports

Figure 92. Process Logic for Determine_System_Mode

```
Frequency:          See DDE for behavior_of_con_Report
Execution Time:
Priority:           Low
Errors Detected:    None
Logic:              See Figure 93
```

Stimulus	Response
received (Vessel_Request = "History_Report_Request")	Report.Report_Type <-- "Weather_History_Report" Report.ASCII_Report <-- read Weather_History_Report from the Report_History data store and convert to ASCII send Report to Set_Outgoing_Radio_Message_Value

Figure 93. Process Logic for Generate_History_Report

App.3.2.21 Generate_Ship_Detailed_Report

```
Requirements:       REQ_Relation_for_con_Report,
                    term_Ship_Detailed_Report
Criteria:           Generate_Ship_Detailed_Report is a REQ process
Inputs:             Vessel_Request message,
                    Buoy_Location data,
                    Air_Temperature data,
                    Water_Temperature data,
                    Wind_Magnitude data,
                    Wind_Direction data
Outputs:            Report message
Frequency:          See DDE for behavior_for_con_Report
Execution Time:
Priority:           Medium
Errors Detected:    None
Logic:              See Figure 94
```

App.3.2.22 Generate_Airplane_Detailed_Report

```
Requirements:       REQ_Relation_for_con_Report,
                    term_Airplane_Detailed_Report
Criteria:           Generate_Airplane_Detailed_Report is a REQ process
Inputs:             Vessel_Request message,
                    Buoy_Location data,
                    Air_Temperature data,
```

Stimulus	Response
received (Vessel_Request = "Ship_Detailed_Report_Request")	Report.Report_Type <-- "Ship_Detailed_Report" Buoy_Location <-- get Buoy_Location read Water_Temperature values from Water_Temperature data store calculate term_Averaged_Water_Temperature read Air_Temperature values from Air_Temperature data store calculate term_Averaged_Air_Temperature read Wind_Direction values from Wind_Direction data store calculate term_Averaged_Wind_Direction read Wind_Magnitude from Wind_Magnitude data store calculate term_Averaged_Wind_Magnitude term_Ship_Detailed_Report <-- (Buoy_Location, term_Averaged_Water_Temperature, term_Averaged_Air_Temperature, term_Averaged_Wind_Direction, term_Averaged_Wind_Magnitude) Report.ASCII_Report <-- ASCII(term_Ship_Detailed_Report) send Report to Set_Outgoing_Radio_Message_Value

Figure 94. Process Logic for Generate_Ship_Detailed_Report

```
                    Water_Temperature data,
                    Wind_Magnitude data,
                    Wind_Direction data
Outputs:            Report message
Frequency:          See DDE for behavior_for_con_Report
Execution Time:
Priority:           Medium
Errors Detected:    None
Logic:              See Figure 95
```

Stimulus	Response
received (Vessel_Request = "Airplane_Detailed_Report_Request")	Report.Report_Type <-- "Airplane_Detailed_Report" -- generate the same report as generated by -- Generate_Ship_Detailed_Report send Report to Set_Outgoing_Radio_Message_Value

Figure 95. Process Logic for Generate_Airplane_Detailed_Report

App.3.2.23 Send_Outgoing_Radio_Message

```
Requirements:       out_Outgoing_Radio_Message,
                    OUT_Relation_for_con_Report
Criteria:           Send_Outgoing_Radio_Message is an OUTs process
Inputs:             Outgoing_Radio_Message message
Outputs:            Outgoing_Radio_Message data
Frequency:          See DDE for behavior_of_con_Report
Execution Time:
Priority:           High
Errors Detected:    None
Logic:              See Figure 96
```

App.3.2.24 Control_Light_Switch

```
Requirements:       out_Light_Switch,
                    OUT_Relation_for_con_Red_Light
```

Stimulus	Response
received Outgoing_Radio_Message	write Outgoing_Radio_Message to RegisterG

Figure 96. Process Logic for Send_Outgoing_Radio_Message

```
Criteria:          Control_Light_Switch is an OUTs process
Inputs:            Light_Switch message
Outputs:           Light_Switch data
Frequency:         See DDE for behavior_of_con_Red_Light
Execution Time:
Priority:          Medium
Errors Detected:   None
Logic:             See Figure 97
```

Stimulus	Response
received Light_Switch	write Light_Switch to RegisterH

Figure 97. Process Logic for Control_Light_Switch

App.3.2.25 Determine_Reset_SOS

```
Requirements:      IN_Relation_for_mon_Reset_SOS,
Criteria:          Determine_Reset_SOS is an INt process
Inputs:            Incoming_Radio_Message message
Outputs:           Reset_SOS message
Frequency:         see DDE for event_Incoming_Radio_Message_6
Execution Time:
Priority:          Medium
Errors Detected:   None
Logic:             See Figure 98
```

Stimulus	Response
received (Incoming_Radio_Message = 6)	Reset_SOS <-- "True" send Reset_SOS to Determine_System_Mode

Figure 98. Process Logic for Determine_Reset_SOS

App.3.2.26 Determine_Light_Command

```
Requirements:      IN_Relation_for_mon_Light_Command,
Criteria:          Determine_Light_Command is an INt process
Inputs:            Incoming_Radio_Message message
Outputs:           Light_Command message
Frequency:         see DDEs for event_Incoming_Radio_Message_1, and
                   event_Incoming_Radio_Message_2
Execution Time:
Priority:          Medium
```

```
Errors Detected:   None
Logic:             See Figure 99
```

Stimulus	Response
received (Incoming_Radio_Message = 1)	Light_Command <-- "Red_Light_On" send Light_Command to Process_Red_Light_Request
received (Incoming_Radio_Message = 2)	Light_Command <-- "Red_Light_Off" send Light_Command to Process_Red_Light_Request

Figure 99. Process Logic for Determine_Light_Command

App.3.2.27 Determine_Omega_Error

```
Requirements:      IN_Relation_for_mon_Omega_Error,
Criteria:          Determine_Omega_Error is an INt process
Inputs:            Location_Correction_Data message
Outputs:           Omega_Error message
Frequency:         see DDE for event_Incoming_Radio_Message_7
Execution Time:
Priority:          Medium
Errors Detected:   None
Logic:             See Figure 100
```

Stimulus	Response
received Location_Correction_Data	Omega_Error <-- (Lat_Offset <= Location_Correction_Data.u, Lon_Offset <= Location_Correction_Data.l) send Omega_Error to Determine_Buoy_Location

Figure 100. Process Logic for Determine_Omega_Error

App.3.3 Process Architecture Diagram

Figure 101 shows the process architecture diagram that resulted from applying the ADARTS process clustering criteria to the processes on the initial process architecture diagram in Figure 73. The process behavior specifications in Section App.3.4 describe how and when the criteria were applied to obtain the process architecture as illustrated in Figure 101.

App.3.4 Process Behavior Specifications

This section contains the process behavior specifications associated with the processes on the process architecture diagram in Figure 101. Note that **teamwork** SEMs and PATs were used to specify the logic of processes in stimulus/response form. Sections App.3.4.1 through App.3.4.8 each contain a process behavior specification corresponding to one of the processes on the process architecture diagram.

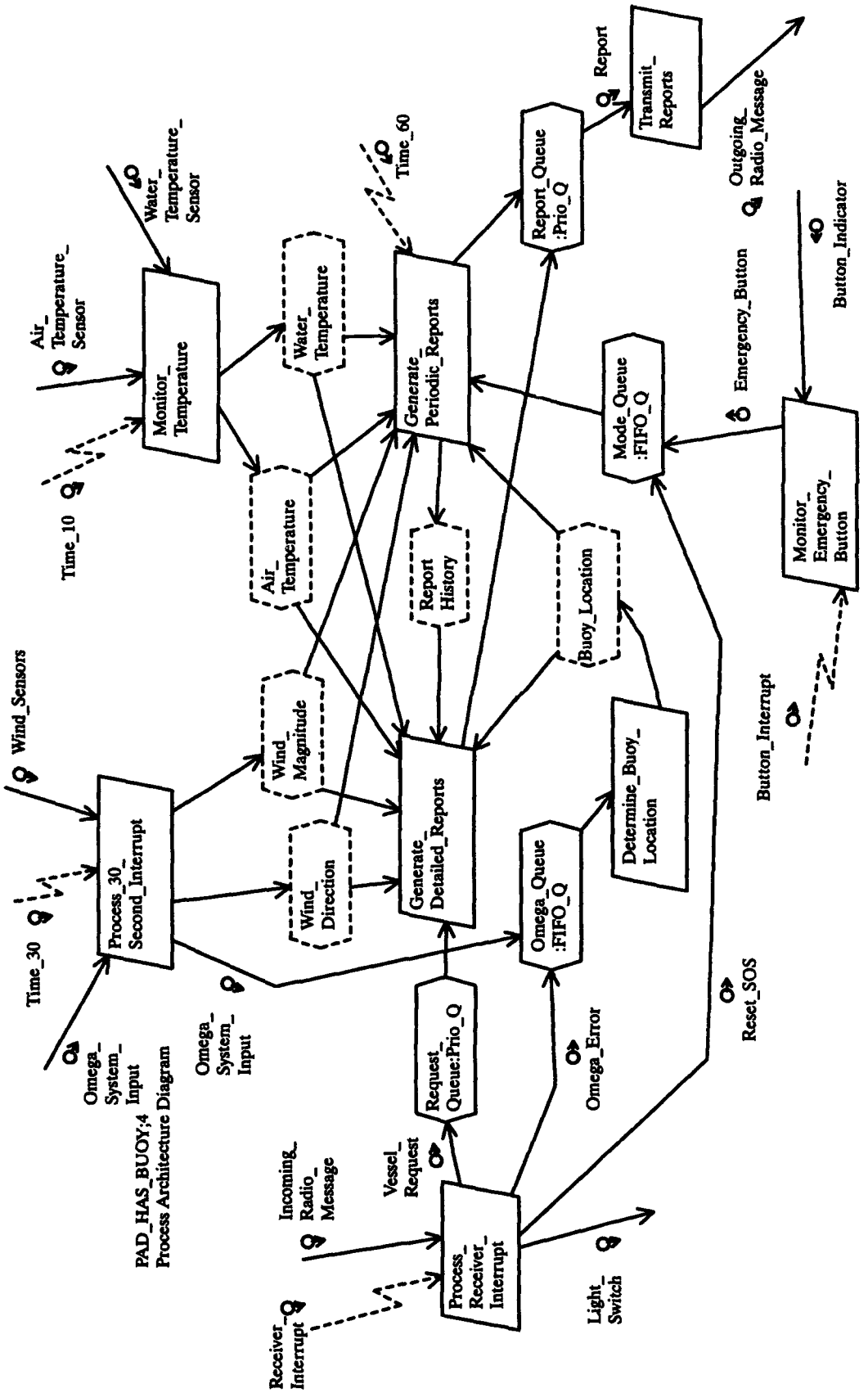


Figure 101. HAS Buoy Process Architecture Diagram

App.3.4.1 Process_30_Second_Interrupt

Requirements:	IN_Relation_for_mon_Wind, and term_Wind_Vector, in_Wind_Sensors, in_Omega_System_Input, IN_Relation_for_mon_Buoy_Location
Criteria:	Determine_Wind_Direction and Determine_Wind_Magnitude (both INt processes) were clustered based on asynchronous temporal cohesion - they were both activated by the receipt of Wind_Sensors. The resulting process was clustered with Monitor_Wind_Sensors (an INs process) based on sequential cohesion. This process was then clustered with Monitor_Omega_System_Input (INs process) based on periodic temporal cohesion - they were both activated at 30 second intervals.
Inputs:	Wind_Sensors data, Omega_System_Input data, Time_30 event
Outputs:	Wind_Direction data, Wind_Magnitude data, Omega_System_Input message
Frequency:	every 30 seconds
Execution Time:	
Priority:	Medium
Errors Detected:	Wind sensors device error
Logic:	See Figure 102

App.3.4.2 Monitor_Temperature

Requirements:	IN_Relation_for_mon_Air_Temperature, in_Air_Temperature_Sensor, IN_Relation_for_mon_Water_Temperature, in_Water_Temperature_Sensor
Criteria:	First, the entity modeling, INs processes defined by Monitor_Air_Temperature_Sensors_Multiple were clustered into a single process using entity process inversion. The resulting was clustered with Determine_Air_Temperature (an INt process) based on sequential cohesion. Then, the entity modeling, INs process defined by Monitor_Air_Temperature_Sensors_Multiple were also clustered into a single process using entity process inversion. This resulting was clustered with Determine_Air_Temperature (an INt process) based on sequential cohesion. Finally, the two resulting processes were clustered based on periodic temporal cohesion - they were both activated at 10 second intervals.
Inputs:	Air_Temperature_Sensor data, Water_Temperature_Sensor data, Time_10 event
Outputs:	Air_Temperature data, Water_Temperature

Stimulus	Response

when Time_30	North <-- Read(RegiserC1)
	South <--Read(RegisterC2)
	East <-- Read(RegisterC3)
	West <-- Read(RegisterC4)
	Omega_System_Input <-- read(RegisterD) where
	(Latitude.Degrees <-- RegisterD.Bytes_1&2
	Latitude.Minutes <-- RegisterD.Byte_3
	Latitude.Seconds <-- RegisterD.Byte_4
	Latitude.Hundredths <-- RegisterD.Byte_5
	Longitude.Degrees <--RegisterD.Bytes_6&7
	Longitude.Minutes <-- RegisterD.Byte_8
	Longitude.Seconds <-- RegisterD.Byte_9
	Longitude.Hundredths <-- RegisterD.Byte_10)
	send Omega_System_Input to Omega_Queue
	if ((North > 0) AND (South > 0)) OR ((East > 0) AND (West > 0)) then
	there is a device error
	else
	Wind_Direction <--
	if (West > 0) AND (South > 0) then use ROUND(INVTAN(West / South) + 180)
	elsif (West > 0) AND (North > 0) then use ROUND(INVTAN(North / West) + 270)
	elsif (East > 0) AND (North > 0) then use ROUND(INVTAN(East / North))
	elsif (East > 0) AND (South > 0) then use ROUND(INVTAN(South / East) + 90)
	elsif (West <= 0) AND (East <= 0) AND (South > 0) then use 180
	elsif (West <= 0) AND (East <= 0) AND (South <= 0) then use 0
	write Wind_Direction to the Wind_Direction data store
	Wind_Magnitude <--
	if (West > 0) AND (South > 0) use SQRT(SQUARE(South) + SQUARE(West))
	elsif (West > 0) AND (North > 0) use SQRT(SQUARE(North) + SQUARE(West))
	elsif (East > 0) AND (North > 0) use SQRT(SQUARE(North) + SQUARE(East))
	elsif (East > 0) and (South > 0) use SQRT(SQUARE(South) + SQUARE(East))
	elsif (West <= 0) AND (East <= 0) AND (South > 0) use South
	elsif (West <= 0) AND (East <= 0) AND (North > 0) use North
	elsif (North <= 0) AND (South <= 0) AND (East > 0) use East
	elsif (North <= 0) AND (South <= 0) AND (West > 0) use West
	elsif (North <= 0) AND (South <= 0) AND (East <= 0) AND (West <= 0) use 0
	write Wind_Magnitude to Wind_Magnitude data store

Figure 102. Process Logic for Process_30_Second_Interrupt

```
Frequency:          every 10 seconds
Execution Time:
Priority:           Medium
Errors Detected:    None
Logic:              See Figure 103
```

App.3.4.3 Determine_Buoy_Location

```
Requirements:       IN_Relation_for_mon_Buoy_Location,
Criteria:           Determine_Buoy_Location is an INt process
Inputs:             Omega_System_Input message,
                    Omega_Error message
Outputs:            Buoy_Location data
Frequency:          once per 30 seconds for Omega_System_Input,
                    see DDE event_Incoming_Radio_Message_7
Execution Time:
Priority:           Medium
Errors Detected:    None
Logic:              See Figure 104
```

Stimulus	Response
when Time_10	Air_Temperature_Sensor <-- Read (RegisterB1) Air_Temperature <-- 200 x (Air_Temperature_Sensor + 128) / 256 - 100 write Air_Temperature to the Air_Temperature data store Air_Temperature_Sensor <-- Read (RegisterB2) Air_Temperature <-- 200 x (Air_Temperature_Sensor + 128) / 256 - 100 write Air_Temperature to the Air_Temperature data store Water_Temperature_Sensor <-- Read (RegisterA1) Water_Temperature <-- (104 x Water_Temperature_Sensor_1) / 255 - 4 write Water_Temperature to the Water_Temperature data store Water_Temperature_Sensor <-- Read (RegisterA2) Water_Temperature <-- (104 x Water_Temperature_Sensor_1) / 255 - 4 write Water_Temperature to the Water_Temperature data store

Figure 103. Process Logic for Monitor_Temperature

Stimulus	Response
received Omega_System_Input or Omega_Error	get next entry from Omega_Queue if entry is Omega_System_Input then Buoy_Location.Latitude <= (Degrees <= MAX(<Latitude>Omega_System_Input.Bytes_1&2, 359), Minutes <= MAX(<Latitude>Omega_System_Input.Byte_3, 59), Seconds <= MAX(<Latitude>Omega_System_Input.Byte_4, 59) + MAX(<Latitude>Omega_System_Input.Byte_5, 99) / 100), Buoy_Location.Longitude <= (Degrees <= MAX(<Longitude>Omega_System_Input.Bytes_1&2, 359), Minutes <= MAX(<Longitude>Omega_System_Input.Byte_3, 59), Seconds <= MAX(<Longitude>Omega_System_Input.Byte_4, 59) + MAX(<Longitude>Omega_System_Input.Byte_5, 99) / 100) Buoy_Location <-- Adjust_for_Error (Buoy_Location, Omega_Error) write Buoy_Location to Buoy_Location data store elsif entry is Omgea_Error then store Omega_Error locally for future calculations of Buoy_Location

Figure 104. Process Logic for Determine_Buoy_Location

App.3.4.4 Generate_Periodic_Reports

Requirements:	REQ_Relation_for_con_Report, Mode_Machine_for_System_Mode, term_SOS_Report, term_Wind_and_Temperature_Report
Criteria:	Generate_Periodic_Reports (a REQ process) was clustered with Determine_System_Mode (a mode process) based on functional cohesion.
Inputs:	Mode_Change message (Reset_SOS or Emergency_Button) Buoy_Location data, Air_Temperature data, Water_Temperature data, Wind_Magnitude data, Wind_Direction data, Time_60 event
Outputs:	Report message
Frequency:	See DDE for behavior_of_con_Report
Execution Time:	
Priority:	High

Errors Detected: None
Logic: See Figure 105

Stimulus	Response
when Time_60	if (System_Mode = "mode_SOS") then Report.Report_Type <-- "SOS_Report" SOS_Report <-- read Buoy_Location data store Report.ASCII_Report <-- ASCII(SOS_Report) send Report to Report_Queue elsif (System_Mode = "mode_Normal") then Report.Report_Type <-- "Wind_and_Temperature_Report" read Water_Temperature values from Water_Temperature data store calculate term_Averaged_Water_Temperature read Air_Temperature values from Air_Temperature data store calculate term_Averaged_Air_Temperature read Wind_Direction values from Wind_Direction data store calculate term_Averaged_Wind_Direction read Wind_Magnitude from Wind_Magnitude data store calculate term_Averaged_Wind_Magniutde Wind_and_Temperature_Report <-- (term_Averaged_Water_Temperature, term_Averaged_Air_Temperature, term_Averaged_Wind_Direction, term_Averaged_Wind_Magnitude) Report.ASCII_Report = ASCII(Wind_and_Temperature_Report) send Report to the Report_Queue write Report to the Report_History data store

received Mode_Change	if (Mode_Change = [Emergency_Button = "Pressed"]) then
	if (System_Mode = "mode_Normal") then
	System_Mode <-- "mode_SOS"
	if (Mode_Change = [Reset_SOS = "True"]) then
	if (System_Mode = "mode_SOS") then
	System_Mode <-- "mode_Normal"

Figure 105. Process Logic for Generate_Periodic_Reports

App.3.4.5 Process_Receiver_Interrupt

Requirements: in_Incoming_Radio_Message,
in_Location_Correction_Data,
IN_Relation_for_mon_Reset_SOS,
IN_Relation_for_mon_Omega_Error,
IN_Relation_for_mon_Vessel_Request,
IN_Relation_for_mon_Light_Command,
REQ_Relation_for_con_Red_Light,
OUT_Relation_for_con_Red_Light,
out_Light_Switch

Criteria: Monitor_Incoming_Radio_Messages (an INs process) and Monitor_Location_Correction_Data (an INs process) were clustered based on asynchronous temporal cohesion - they were both activated by Receiver_Interrupt. The resulting process was clustered with Determine_Omega_Error (an INt process) based on sequential cohesion. Again, the resulting process was clustered with the following processes, all based on sequential cohesion: Determine_Light_Command (an INt process), Process_Red_Light_Request (a REQ process), Set_Light_Switch_Value (an OUTt process), and

```
                    Control_Light_Switch (an OUTs process). Sequential
                    cohesion was then applied to complete the clustering with
                    Determine_Reset_SOS (an INt process) and
                    Determine_Vessel_Request (an INt process)..
Inputs:             Incoming_Radio_Message data,
                    Receiver_Interrupt event
Outputs:            Incoming_Radio_Message message,
                    Omega_Error message,
                    Reset_SOS message,
                    Vessel_Request message,
                    Light_Switch data
Frequency:          see DDEs for event_Incoming_Radio_Message_1,
                    event_Incoming_Radio_Message_2,
                    event_Incoming_Radio_Message_3,
                    event_Incoming_Radio_Message_4,
                    event_Incoming_Radio_Message_5,
                    event_Incoming_Radio_Message_6,
                    event_Incoming_Radio_Message_7, and
                    behavior_of_con_Red_Light,
Execution Time:
Priority:           Medium
Errors Detected:    None
Logic:              See Figure 106
```

Stimulus	Response
received Receiver_Interrupt	Read (RegisterF) case RegisterF.Byte_1 is when 16#01# => Light_Switch <-- 2#1xxxxxxx# write Light_Switch to RegisterH when 16#02# => Light_Switch <-- 2#0xxxxxxx# write Light_Switch to RegisterH when 16#03 => Vessel_Request <-- "History_Report_Request" send Vessel_Request to Request_Queue when 16#04# => Vessel_Request <-- "Airplane_Detailed_Report_Request" send Vessel_Request to Request_Queue when 16#05# => Vessel_Request <-- "Ship_Detailed_Report_Request" send Vessel_Request to Request_Queue when 16#06 => Reset_SOS <-- "True" send Reset_SOS to Transitions_Comm when 16#07# => Omega_Error <-- (Lat_Offset <= RegisterF.Byte_2, Lon_Offset <= RegisterF.Byte_3) send Omega_Error to Omega_Queue

Figure 106. Process Logic for Process_Receiver_Interrupt

App.3.4.6 Monitor_Emergency_Button

```
Requirements:  in_Button_Indicator,
     IN_Relation_for_mon_Emergency_Button,
```

Criteria: Monitor_Button_Indicator (an INs process) and Determine_Emergency_Button (an INt process) were clustered based on sequential cohesion.
Inputs: Button_Indicator data, Button_Interrupt event
Outputs: Emergency_Button message
Frequency: see DDEs for event_Emergency_Button_Pressed and event_Emergency_Button_Released
Execution Time:
Priority: Medium
Errors Detected: None
Logic: See Figure 107

Stimulus	Response

received Button_Interrupt	read Button_Indicator from RegisterE
	if (Button_Indicator = 2#1xxxxxxx#) then
	Emergency_Button <-- "Pressed"
	send Emergency_Button to Transitions_Comm
	else
	Emergency_Button <-- "Released"

Figure 107. Process Logic for Monitor_Emergency_Button

App.3.4.7 Transmit_Reports

Requirements: OUT_Relation_for_con_Report, out_Outgoing_Radio_Message
Criteria: Set_Outgoing_Radio_Message_Value (an OUTt process) and Send_Outgoing_Radio_Message (an OUTs process) were clustered based on sequential cohesion and renamed Transmit_Reports.
Inputs: Report message
Outputs: Outgoing_Radio_Message data
Frequency: See DDE for behavior_of_con_Report
Execution Time:
Priority: High
Errors Detected: None
Logic: See Figure 108 (Note: Prioritization of reports must be enforced.)

App.3.4.8 Generate_Detailed_Reports

Requirements: REQ_Relation_for_con_Report, term_Ship_Detailed_Report, term_Airplane_Detailed_Report, term_Weather_History_Report
Criteria: Generate_Ship_Detailed_Report, Generate_Airplane_Detailed_Report, and Generate_History_Report (all REQ processes) were clustered based on functional cohesion.
Inputs: Vessel_Request message, Buoy_Location data,

Stimulus	Response
received Report	get next Report from Report_Queue case Report.Report_Type is when "SOS_Report" => Page_Count <-- Length (Report.ASCII_Report) / 510 Outgoing_Radio_Message.Report_Code <-- 2#10000001# Outgoing_Radio_Message.Page_Count.Bits_0-3 <-- Page_Count for Iterator in 1 .. Page_Count loop Outgoing_Radio_Message.Page_Count.Bits4-7 <-- Iterator Outgoing_Radio_Message.Bytes_3-512 <-- Report.ASCII_Report(((Page_Number-1) * 510) + 1 .. (Page_Number * 510)) write Outgoing_Radio_Message to RegisterG end for loop when "Wind_and_Temperature_Report" => -- do same as for "SOS_Report" except assign 2#10000010# to --- Outgoing_Radio_Message.Report_Code when "Airplane_Detailed_Report" => -- do same as for "SOS_Report" except assign 2#10000011# to -- Outgoing_Radio_Message.Report_Code when "Ship_Detailed_Report" => -- do same as for "SOS_Report" except assign 2#10000100# to -- Outgoing_Radio_Message.Report_Code when "Weather_History_Report" => -- do same as for "SOS_Report" except assign 2#10000101# to -- Outgoing_Radio_Message.Report_Code

Figure 108. Process Logic for Transmit_Reports

```
                    Air_Temperature data,
                    Water_Temperature data,
                    Wind_Magnitude data,
                    Wind_Direction data,
                    Report_History data
Outputs:            Report message
Frequency:          See DDE for behavior_for_con_Report
Execution Time:
Priority:           Medium
Errors Detected:    None
Logic:              See Figure 109
```

APP.4 CLASS STRUCTURE

This section contains specifications for selected classes derived from the CoRE specification of the HAS Buoy requirements. Behavior is described formally to take advantage of CoRE's precision. However, formal descriptions are not required for class structuring in the ADARTS method.

This section uses the following prefixes in addition to the naming conventions discussed in Section 2.7:

- "param_" identifies a parameter to an operation.
- "result_" identifies the result of an operation.
- "state_" identifies an attribute of the abstract state of a class.

Stimulus	Response

received Vessel_Request	get next Vessel_Request from Request_Queue
	if (Vessel_Request = "History_Report_Request") then
	Report.Report_Type <-- "Weather_History_Report"
	Report.ASCII_Report <-- read Weather_History_Report from
	the Report_History data store and convert to ASCII
	else
	Buoy_Location <-- get Buoy_Location
	read Water_Temperature values from Water_Temperature data store
	calculate term_Averaged_Water_Temperature
	read Air_Temperature values from Air_Temperature data store
	calculate term_Averaged_Air_Temperature
	read Wind_Direction values from Wind_Direction data store
	calculate term_Averaged_Wind_Direction
	read Wind_Magnitude from Wind_Magnitude data store
	calculate term_Averaged_Wind_Magnitude
	Detailed_Report <-- (Buoy_Location,
	term_Averaged_Water_Temperature, term_Averaged_Air_Temperature,
	term_Averaged_Wind_Direction, term_Averaged_Wind_Magnitude)
	Report.ASCII_Report <-- ASCII(Detailed_Report)
	if (Vessel_Request = "Airplane_Detailed_Report_Request") then
	Report.Report_Type <-- "Airplane_Detailed_Report"
	elsif (Vessel_Request = "Ship_Detailed_Report_Request) then
	Report.Report_Type <-- "Ship_Detailed_Report"
	send Report to Report_Queue

Figure 109. Process Logic for Generate_Detailed_Reports

For device interface classes, the abstract state can be the input or output variable associated with the device. In this case, the naming convention is not followed. The name of the input or output variable is used instead.

Each parameter to an operation is associated with the CoRE artifact that the parameter represents (e.g., approximation to monitored variable, term, etc.). It is assumed that the implementation of classes and objects will make use of strong typing, implying a usage constraint prohibiting usage of the wrong type of parameter. Because these usage constraints are so ubiquitous, they (and the associated undesired events) are omitted from the specifications.

App.4.1 Air_Temperature_Sensor Device Interface Class

```
Name:  Air_Temperature_Sensor Device Interface

Abstraction:  Device interface class that defines the interface to the
air temperature sensor, and approximates the value of the Air Temper-
ature monitored variable.

Hidden Information:   Details of operating the air temperature sensor
and approximating the Air Temperature monitored variable.

Anticipated Changes:  None.

Requirements Traceability:

   in_Air_Temperature_Sensor
```

mon_Air_Temperature

In_Relation_for_Air_Temperature

Object(s)

Air_Temperature_Sensor Device Interface object

FORMAL DESCRIPTION

Abstract State: in_Air_Temperature_Sensor

Abbreviations:

Abbreviation	Definition
Valid_Sensor_Input	-128 <= in_Air_Temperature_Sensor <= 127

Invariants: There are no invariants

Initial Value of Abstract State: The value of in_Air_Temperature_Sensor when the system is initiated.

App.4.1.1 Calculate_Air_Temperature Operation

Usage Constraints: None. (IN relation states that a value is always available).

Undesired Events: An undesired event is returned if the usage constraint is violated.

Effects: Each time it is called, this operation retrieves the current value of the Air Temperature Sensor input data item and uses it to approximate the current value of the Air Temperature monitored variable.

Requirements Traceability:

in_Air_Temperature_Sensor

In_Relation_for_Air_Temperature

mon_Air_Temperature

FORMAL DESCRIPTION

Parameters: There are no parameters to this operation.

Results:

result_Air_Temperature (value of ~mon_Air_Temperature)

Abbreviations: See class specification.

Behavior:

Precondition	Postcondition
Valid_Sensor_Input	result_Air_Temperature = ~mon_Air_Temperature as defined by IN' for mon_Air_Temperature Maximum Error: 0.5 degree centigrade
NOT(Valid_Sensor_Input)	ERROR(device failure)

App.4.2 Omega_Navigation_System Device Interface Class

Name: Omega_Navigation_System Device Interface Class

Abstraction: Device interface class encapsulating the Omega Navigation System.

Hidden Information: Details of interfacing with the Omega Navigation System.

Anticipated Changes: Protocol for operating device

Requirements Traceability:

in_Omega_System_Input

Object(s) Omega Navigation System Device Interface object

FORMAL DESCRIPTION

Abstract State: in_Omega_System_Input

Abbreviations: None.

Invariants: There are no invariants

Initial Value of Abstract State: The value of in_Omega_System_Input when the system is initiated.

App.4.2.1 Get_Omega_Input Operation

Usage Constraints: None.

Undesired Events: None.

Effects: The current value of in_Omega_System_Input is returned.

Requirements Traceability:

in_Omega_System_Input

FORMAL DESCRIPTION

Parameters: None.

Results:

result_Omega_System_Input (value of in_Omega_System_Input)

Abbreviations: See class specification.

Behavior:

Precondition	Postcondition
TRUE	result_Omega_System_Input = in_Omega_System_Input

App.4.3 Water_Temperature_Sensor Device Interface Class

Name: Water_Temperature_Sensor Device Interface

Abstraction: Device interface class that defines the interface to the water temperature sensor.

Hidden Information: Details of interfacing with a water temperature sensor.

Anticipated Changes: The current version of the Buoy has a single sensor for water temperature. In the future, the number of water temperatures could change. For this reason, the conversion of the Water Temperature Sensor input value to an approximation of the Water Temperature monitored variable is encapsulated in a different class.

Requirements Traceability:

in_Water_Temperature_Sensor

Object(s) Water_Temperature_Sensor Device Interface object.

FORMAL DESCRIPTION

Abstract State: in_Water_Temperature_Sensor

Abbreviations:

Abbreviation	Definition
Valid_Sensor_Input	-128 <= in_Water_Temperature_Sensor <= 127

Invariants: There are no invariants

Initial Value of Abstract State: The value of in_Air_Temperature_Sensor when the system is initiated.

App.4.3.1 Read_Water_Temperature_Sensor Operation

Usage Constraints: None (IN relation states that a value is always available).

Undesired Events: An error is returned if the condition

-128 <= in_Water_Temperature_Sensor <= 127

does not hold.

Effects: The current Water Temperature Sensor value is returned.

Requirements Traceability:

in_Water_Temperature_Sensor

FORMAL DESCRIPTION

Parameters: There are no parameters.

Results:

result_Water_Temperature_Sensor_Input (value of in_Water_Temperature_Sensor)

Abbreviations: See class specification

Behavior:

Precondition	Postcondition
Valid_Sensor_Input	result_Water_Temperature_Sensor_Input= in_Water_Temperature_Sensor Maximum Error: 0
NOT(Valid_Sensor_Input)	ERROR(device failure)

App.4.4 Wind_Sensor Device Interface Class

Name: Wind_Sensor Device Interface class

Abstraction: This class abstracts the wind sensor devices.

Hidden Information: Details of reading from the wind sensors

Anticipated Changes: The number of wind sensor devices may change.

Requirements Traceability:

in_Wind_Sensors

Object(s):

North_Wind_Sensor object

South_Wind_Sensor object

East_Wind_Sensor object

West_Wind_Sensor object

FORMAL DESCRIPTION

Abstract State: <X>Sensor (Input variable returned by device—see abbreviations)[6].

Abbreviations:

Abbreviation	Definition
<X>	<North> for North Wind Sensor object <South> for South Wind Sensor object <East> for East Wind Sensor object <West> for West Wind Sensor object
Valid_Sensor_Input	-128 <= <X>Sensors <= 127

Invariants: There are no invariants

Initial Value of Abstract State: The value available from the corresponding wind sensor when the system is initiated.

App.4.4.1 Read_Wind_Sensor_Input Operation

Usage Constraints: None

Undesired Events: An error is returned of the wind sensor input value is not in the range 0 to 255 inclusive.

Effects: The current wind sensor value for the wind sensor is returned.

Requirements Traceability:

6. There are four objects derived from this class. The abbreviation <X> serves to distinguish the abstract state of different objects.

in_Wind_Sensors

FORMAL DESCRIPTION

Parameters: None.

Results:

result_Wind_Sensor_Value (value of <X>Sensor).

Abbreviations:

Behavior:

Precondition	Postcondition
Valid_Sensor_Input	Wind_Sensor_Value=<X>Sensors Maximum Error: 0
NOT(Valid_Sensor_Input)	ERROR(device failure)

App.4.5 ASCII_Report Data Abstraction Class

Name: ASCII_Report Data Abstraction Class

Abstraction: Data abstraction class which hides the internal structure of an ASCII report.

Hidden Information: Internal structure of the report.

Anticipated Changes: No changes anticipated at this point.

Requirements Traceability:

con_Report

REQ Relation for con_Report

OUT Relation for con_Report

Object(s) ASCII_Report object

FORMAL DESCRIPTION

Abstract State:

state_ASCII_Report (value of ~con_Report -- a sequence of ASCII characters)

state_Next_Page (Natural number–that indicates which page is to be returned by the Get_Next_Page operation.

state_Pages_Remaining (Boolean value indicating that some pages in the current report are yet to be returned by the Get_Next_Page operation).

Abbreviations:

Abbreviation	Definition
Max_Number_Pages	20 pages
Page_Length	1024 ASCII characters

Invariants: There are no invariants

LENGTH(state_ASCII_Report) <= Max_Number_Pages * Page_Length

Initial Value of Abstract State:

state_Pages_Remaining=FALSE

App.4.5.1 Set_Report Operation

Usage Constraints: After successfully calling this operation, the Get_Next_Page operation must be called once for each page of the report. Also, the length of the report must not exceed the storage capacity of the object derived from this class.

Undesired Events: An error is returned if this operation is called before all pages of the previous report have been returned, or if the parameter contains too many characters.

Effects: The next report to be transmitted is recorded internally, and the first page is available from the Get_Next_Page operation.

Requirements Traceability:

con_Report

REQ Relation for con_Report

OUT Relation for con_Report

FORMAL DESCRIPTION

Parameters:

param_ASCII_Report (value of ~con_Report)

Results: No result is returned.

Abbreviations: See class specification.

Behavior:

Precondition	Postcondition
state_Pages_Remaining=FALSE AND LENGTH(param_ASCII_Report) < Max_Report_Length	state_ASCII_Report =param_ASCII_Report AND state_Next_Page=1 AND state_Pages_Remaining=TRUE
state_Pages_Remaining=TRUE	ERROR(Attempt to overwrite previous report)
LENGTH(param_ASCII_Report) < Max_Report_Length	ERROR(Report too long)

App.4.5.2 Get_Next_Page Operation

Usage Constraints: After calling the Set_Report operation, this operation should be called until all pages of the report have been returned. This operation should not be called again until after the next successful call to Set_Report.

Undesired Events: An error is returned if the usage constraint on sequencing is not observed.

Effects: The next page of the current report is returned. The first page is returned if this is the first call following a call to Set_Report.

Requirements Traceability:

con_Report

REQ Relation for con_Report

OUT Relation for con_Report

FORMAL DESCRIPTION

Parameters: There are no parameters.

Results:

result_Report_Page (value: a sequence up to Page_Length ASCII characters)

result_Last_Page (boolean value—TRUE indicates that result_Report_Page contains the last page of the current report.

Abbreviations: See class specification.

Abbreviation	Definition
Low_Index	Page_Length*(state_Next_Page-1)+1
High_Index	MAX(Page_Length*state_Next_Page, LENGTH(state_ASCII_Report))

Abbreviation	Definition
On_Last_Page	High_Index=LENGTH(State_ASCII_Report)
Page_Returned	state_ASCII_Report(Low_Index..High_Index)

Behavior:

Precondition	Postcondition
state_Pages_Remaining	result_Report_Page=Page_Returned AND result_Last_Page=On_Last_Page AND state_Pages_Remaining= NOT(On_Last_Page)
NOT(state_Pages_Remaining)	ERROR(No more pages)

App.4.6 Buoy_Location Data Abstraction Class

Name: Buoy_Location

Abstraction: Data abstraction class encapsulating an approximation of the current location of the buoy.

Hidden Information: Internal representation of the approximation.

Anticipated Changes: Precision of the approximation.

Requirements Traceability:

mon_Buoy_Location

Object(s)

Buoy_Location object

FORMAL DESCRIPTION

Abstract State:

state_Latitude (value of <Latitude>~mon_Buoy_Location)

state_Longitude (value of <Longitude>~mon_Buoy_Location)

state_Latitude_Defined (Boolean value—TRUE if Set_Latitude operation called at least once).

state_Longitude_Defined (Boolean value—TRUE if Set_Longitude operation called at least once).

Abbreviations: None.

Invariants: None.

Initial Value of Abstract State:

state_Latitude_Defined=FALSE

state_Longitude_Defined=FALSE

Operation(s): [7]

Set Latitude[8] Read Latitude

Set Longitude Read Longitude

App.4.7 SOS_Report Data Abstraction Class

Name: SOS_Report Data Abstraction Class

Abstraction: Data abstraction class which hides the format of ~con_Report while in mode_SOS (i.e., the report transmitted every 60 seconds when the buoy is in SOS mode). The report contains a field which identifies it as an SOS report and the current location (latitude and longitude) of the buoy.

Hidden Information: Format of the report.

Anticipated Changes: Format of the report.

Requirements Traceability:

con_Report

Object(s)

SOS_Report object

FORMAL DESCRIPTION

Abstract State:

state_Latitude (value of <Latitude>~mon_Buoy_Location)

state_Longitude (value of <Longitude>~mon_Buoy_Location)

state_Latitude_Defined (TRUE if the Set_Latitude Operation has been called at least once).

state_Longitude_Defined (TRUE if the Set_Longitude Operation has been called at least once).

7. Descriptions are omitted for brevity.
8. This is similar to the Set_Latitude Operation on the SOS_Report Data Abstraction Class.

Abbreviations: None.

Invariants: There are no invariants

Initial Value of Abstract State:

state_Latitude_Defined=FALSE

state_Longitude_Defined=FALSE

App.4.7.1 Set_Latitude Operation[9]

Usage Constraints: None.

Undesired Events: None.

Effects: The parameter to this operation is recorded in the latitude field of the SOS report.[10]

Requirements Traceability:

con_Report

FORMAL DESCRIPTION

Parameters:

param_Current_Latitude (value of <Latitude>~mon_Buoy_Location).

Results: No result is returned.

Abbreviations: See class specification.

Behavior:

Precondition	Postcondition
TRUE	state_Latitude=param_Current_Latitude state_Latitude_Defined=TRUE

App.4.7.2 ASCII_Format Operation

Usage Constraints: The buoy location must be defined before this operation is called.

Undesired Events: An error is returned if the buoy location is not defined.

9. The Set_Longitude operation is similar and is omitted for brevity.
10. The SOS report also has a field that identifies the type of the report. Because that field is constant, there is no need for an operation to record it.

Effects: An ASCII string containing the current buoy location and a field identifying an SOS report is returned.

Requirements Traceability:

con_Report

REQ_Relation_for_con_Report

FORMAL DESCRIPTION

Parameters: None.

Results:

result_ASCII_Report (value: of ~con_Report.ASCII_Report -- a sequence of ASCII characters)

Abbreviations: See class specification.

Abbreviation	Definition
Location_Defined	state_Latitude_Defined AND state_Longitude_Defined

Behavior:

Precondition	Postcondition
Location_Defined	ASCII_Report = ASCII(state_Latitude) + ASCII(state_Longitude)
NOT(Location_Defined)	ERROR(Undefined Location)

App.4.8 AIR_TEMPERATURE_READINGS COLLECTION CLASS

Name: Air_Temperature_Readings Collection

Abstraction: Data collection class which stores a set of up to six air temperature.

Hidden Information: Method of representing and iterating over the sequence

Anticipated Changes:

Internal representation of the collection

Algorithms for averaging and modifying the collection.

Requirements Traceability:

term_Averaged_Air_Temperature

Object(s)

Air_Temperature_Readings object

FORMAL DESCRIPTION

Abstract State:

state_Collection (Value: A set of up to six elements. The elements are taken from the same domain as mon_Air_Temperature)

Abbreviations: None.

Invariants:

SIZE(state_Collection)$\leq$6

Initial Value of Abstract State: {}

App.4.8.1 Record_Air_Temperature Operation

Usage Constraints: None.

Undesired Events: None.

Effects: This operation adds an air temperature reading to the collection. If the collection is already full, the oldest value is removed to make room for the value to be added.

Requirements Traceability:

term_Averaged_Air_Temperature

FORMAL DESCRIPTION

Parameters:

param_Value -- Value to be added

Results: No result is returned.

Abbreviations: See class specification.

Behavior:

Precondition	Postcondition
SIZE(state_Collection)<6	Updated_state_Collection= state_Collection UNION {param_Value}
SIZE(state_Collection)=6	Updated_state_Collection= state_Collection - OLDEST(state_Collection) UNION {param_Value}

App.4.8.2 Compute_Averaged_Air_Temperature Operation

Usage Constraints: In the requirements specification, term_Averaged_Air_Temperature is defined as an average of six air temperature values, implying that the collection must be full before this operation can be invoked.

Undesired Events: An error is returned if there are fewer than Max Size elements in the collection.

Effects: The arithmetic average of the collection is returned.

Requirements Traceability:

term_Averaged_Air_Temperature

FORMAL DESCRIPTION

Parameters: None.

Results:

result_Averaged_Air_Temperature (value of ~term_Averaged_Air_Temperature)

Abbreviations: None.

Behavior:

Precondition	Postcondition
SIZE(state_Collection)=6	result_Averaged_Air_Temperature = ROUND[SUM(state_Collection)/6] Maximum Error: 1 degree centigrade
SIZE(state_Collection)<6	Error(Insufficient Data)

App.4.9 System_Mode State Transition Class

Name: System_Mode

Abstraction: State transition class which encapsulates Mode_Class_for_mode_System_Mode

Hidden Information: The modes of the mode machine and the transitions between them.

Anticipated Changes: Additional modes and transitions may be added.

Requirements Traceability:

Mode_Class_for_mode_System_Mode

event_Emergency_Button_Pressed

event_Reset_SOS

Object(s)

System_Mode State Transition Object

FORMAL DESCRIPTION

Abstract State:

state_System_Mode (value of ~mode_System_Mode)

Abbreviations: None.

Invariants: There are no invariants

Initial Value of Abstract State: state_System_Mode=mode_Normal

App.4.9.1 Emergency_Button_Pressed Operation

Usage Constraints: None.

Undesired Events: None.

Effects: The value of the abstract state is Emergency.

Requirements Traceability:

event_Emergency_Button_Pressed

FORMAL DESCRIPTION

Parameters: None.

Results: No result is returned.

Abbreviations: See class specification.

Behavior:

Precondition	Postcondition
state_System_Mode=Normal OR state_System_Mode=Emergency	state_System_Mode=Emergency

App.4.9.2 Reset_SOS Operation

Usage Constraints: None.

Undesired Events: None.

Effects: The value of the abstract state is Normal.

Requirements Traceability:

event_Reset_SOS

FORMAL DESCRIPTION

Parameters: None.

Results: No result is returned.

Abbreviations: See class specification.

Behavior:

Precondition	Postcondition
state_System_Mode=Normal OR state_System_Mode=Emergency	state_System_Mode=Normal

App.4.9.3 Current_Mode Operation

Usage Constraints: None.

Undesired Events: None.

Effects: The current value of the abstract state is returned.

Requirements Traceability:

REQ_Relation_for_con_Report

FORMAL DESCRIPTION

Parameters: None.

Results:

result_Current_Mode (value of ~mode_System_Mode)

Abbreviations: See class specification.

Behavior:

Precondition	Postcondition
TRUE	result_Current_Mode=state_System_Mode

App.4.10 Buoy_Location Computation Class

Name: Buoy_Location Computation Class

Abstraction: Computation class encapsulating the algorithm to derive ~mon_Buoy_Location (i.e., current buoy location) from ~mon_Omega_Error and in_Omega_System_Input

Hidden Information: Details of the algorithm

Anticipated Changes:

Internal representation of intermediate results

Required precision

Requirements Traceability:

IN_Relation_for_mon_Buoy_Location

Object(s)

Buoy_Location Computation object

FORMAL DESCRIPTION

Abstract State: This class has no abstract state.

Abbreviations: None.

Invariants: There are no invariants

Initial Value of Abstract State: Not applicable

App.4.10.1 Estimate_Buoy_Location Operation

Usage Constraints: The sooner this operation is called after retrieving in_Omega_System_Input, the more precise the result will be.

Undesired Events: None.

Effects: The calculated value of ~mon_Buoy_Location is returned.

Requirements Traceability:

IN_Relation_for_mon_Buoy_Location

FORMAL DESCRIPTION

Parameters:

param_Omega_Error (value of ~mon_Omega_Error)

param_Omega_Input (value of in_Omega_System_Input)

Results:

result_Buoy_Location (calculated value of ~mon_Buoy_Location)

Abbreviations: See class specification.

Behavior:

Precondition	Postcondition
TRUE	result_Buoy_Location = param_Omega_Input - param_Omega_Error Maximum Error: 0.01 km

App.4.11 WATER_TEMPERATURE COMPUTATION CLASS

Name: Water_Temperature Computation class

Abstraction: Computation class which encapsulates the algorithm for converting the value of in_Water_Temperature_Sensor to an approximation of mon_Water_Temperature.

Hidden Information: Details of the conversion algorithm.

Anticipated Changes: The current version of the Buoy has a single sensor for water temperature. In the future, the number of water temperatures could change. For this reason, the conversion of in_Water_Temperature_Sensor to an approximation of mon_Water_Temperature is encapsulated in a class separate from the Water_Temperature_Sensor device interface class.

Requirements Traceability:

mon_Water_Temperature

IN_Relation_for_Water_Temperature

Object(s) Water_Temperature Computation object

FORMAL DESCRIPTION

Abstract State: This class has no abstract state.

Abbreviations: None.

Invariants: There are no invariants

Initial Value of Abstract State: Not applicable

App.4.11.1 Calculate_Water_Temperature Operation

Usage Constraints: The value of the parameter to this operation must be in the range [-128 .. 127]

Undesired Events: An error is returned if the usage constraint is violated.

Effects: Given a value of in_Water_Temperature_Sensor, this operation returns an approximation of mon_Water_Temperature.

Requirements Traceability:

mon_Water_Temperature

IN_Relation_for_Water_Temperature

FORMAL DESCRIPTION

Parameters:

param_Water_Temperature_Sensor (value of in_Water_Temperature_Sensor)

Results:

result_Water_Temperature (value of ~mon_Water_Temperature)

Abbreviations:

Abbreviation	Definition
Valid_Parameter	-128 <= param_Water_Temperature_Sensor <= 127

Behavior:

Precondition	Postcondition
Valid_Parameter	result_Water_Temperature= ~mon_Water_Temperature as defined in IN'_for_Mon_Water_Temperature Maximum Error: 0
NOT(Valid_Parameter)	ERROR(Invalid Parameter)

App.4.12 Wind Computation Class

Name: Wind Computation Class

Abstraction: Computation class which encapsulates the algorithms for deriving approximations of wind direction and magnitude given the values read from the North, South, East, and West wind sensors. The algorithms are grouped into a single class because the functions they compute have several mathematical terms in common, implying that the algorithms will change together.

Anticipated Changes: Change to one or both algorithms.

Requirements Traceability:

IN_Relation_for_Wind

term_Wind_Vector

Object(s) Wind Computation object

FORMAL DESCRIPTION

Abstract State: This class has no abstract state.

Abbreviations: The abbreviations below are defined in terms of parameters to the two operations exported by this class.

Abbreviation	Definition
Sensor_Values_Non_Negative	param_Wind_Sensors.North >= 0 AND param_Wind_Sensors.South >= 0 AND param_Wind_Sensors.East >= 0 AND param_Wind_Sensors.West >= 0
X_Axis_Values_Consistent	param_Wind_Sensors.East = 0 OR param_Wind_Sensors.West = 0
Y_Axis_Values_Consistent	param_Wind_Sensors.North = 0 OR param_Wind_Sensors.South = 0
Valid_Input	Sensor_Values_Non_Negative AND X_Axis_Values_Consistent AND Y_Axis_Values_Consistent
Wind_Velocity_X_Axis	IF Wind_Sensors.East>=0 THEN param_Wind_Sensors.East ELSE param_Wind_Sensors.West
Wind_Velocity_Y_Axis	IF Wind_Sensors.North>=0 THEN param_Wind_Sensors.North ELSE param_Wind_Sensors.South

Invariants: There are no invariants

Initial Value of Abstract State: Not applicable

App.4.12.1 Calculate_Wind_Direction Operation

Usage Constraints: The wind sensor readings must be non-negative. The north and south sensor values cannot be positive at the same time. The east and west sensor values cannot be positive at the same time. See the abbreviation Valid_Input in the class specification.

Undesired Events: An error condition is returned if the usage constraint is violated. See Postcondition for NOT (Valid_Input) below.

Effects: Given the four Wind Sensor values read from the four wind sensors, this operation returns an approximation of the Wind Direction monitored variable.

Requirements Traceability:

IN_Relation_for_Wind

term_Wind_Vector

FORMAL DESCRIPTION

Parameters:

param_WindSensor.North

param_WindSensor.South

param_WindSensor.East

param_WindSensor.West

Results:

result_Wind_Direction (value of ~mon_Wind_Direction)

Abbreviations:

Abbreviation	Definition
MAGNITUDE	Calculate_Wind_Magnitude (param_WindSensor.North, param_WindSensor.South, param_WindSensor.East, param_WindSensor.West)
ARC_COS	Trigonometric functions computation class.arc_cos

Behavior:

Precondition	Postcondition
Valid_Input	result_Wind_Direction= ARC_COS(Wind_Velocity_X_Axis/MAGNITUDE) Maximum Error: 1 degree of angle
NOT (Valid_Input)	ERROR (invalid parameters)

App.4.12.2 Calculate_Wind_Magnitude Operation

Name: Calculate_Wind_Magnitude

Usage Constraints: See Calculate_Wind_Direction

Undesired Events: An exception is returned if the usage constraint is violated. See the second postcondition below.

Effects: Given the four Wind Sensor values read from the four wind sensors, this operation returns an approximation of the Wind Magnitude monitored variable.

Requirements Traceability:

IN_Relation_for_Wind

term_Wind_Vector

FORMAL DESCRIPTION

Parameters:

param_WindSensor.North

param_WindSensor.South

param_WindSensor.East

param_WindSensor.West

Results:

result_Wind_Magnitude (Approximation of monitored variable)

Abbreviations: See class specification.

Behavior:

Precondition	Postcondition
Valid_Input	result_Wind_Magnitude = SQRT(Wind_Velocity_X_Axis**2 + Wind_Velocity_Y_Axis**2) Maximum Error: 0.5 knot
NOT (Valid_Input)	Exception (invalid parameters)

App.4.13 TRIGONOMETRIC_FUNCTIONS COMPUTATION CLASS

Name: Trigonometric_Functions Computation Class

Abstraction: Computation class which hides algorithms for common mathematical functions from trigonometric.

Hidden Information: Details of the algorithms

Anticipated Changes:

Precision of the algorithms

Requirements Traceability:

mon_Wind_Magnitude

mon_Wind_Direction

Object(s): Trigonometric_Functions Computation object

FORMAL DESCRIPTION

Abstract State: This class has no abstract state.

Abbreviations: None.

Invariants: There are no invariants

Initial Value of Abstract State. Not applicable

Operation(s): Some of the operations listed are not required by the HAS-Buoy application. However, this class will include a complete set of trigonometric operations so that it can be reused in other applications.

cos arc cos

sin arc sin

tan arc tan

This page intentionally left blank.

LIST OF ABBREVIATIONS AND ACRONYMS

ADARTS	Ada-based Design Approach for Real-Time Systems
CoRE	Consortium Requirements Engineering
DDE	data dictionary entries
FIFO	first-in, first-out
GCD	greatest common divisor
HAS	host-at-sea
IN	input
mph	miles per hour
ms	millisecond
NAT	nature
OUT	output
PAT	process activation table
REQ	required
RTSA	real-time structured analysis
s	stimulus
sec	second
SEM	state-event matrix
t	translation

This page intentionally left blank.

REFERENCES

Cadre Technologies, Inc. 1990 — *Teamwork/SA and teamwork/RT User's Guide.* Providence, Rhode Island: Cadre Technologies, Inc.

Gries, David 1981 — *The Science of Programming.* New York, New York: Springer-Verlag.

Kirk, Richard A., and Fred Wild 1992 — *Using Teamwork Version 4.0 for ADARTS Version 2.0.* Providence, Rhode Island: Cadre Technologies, Inc.

Knuth, Donald E. 1981 — *The Art of Computer Programming Vol. 2. Seminumerical Algorithms.* Second Edition. Reading, Massachusetts: Addison-Wesley.

Naval Research Laboratory 1980 — *Software Engineering Principles.* Washington, D.C.: Naval Research Laboratory.

Software Productivity Consortium 1991 — *ADARTS Guidebook.,* SPC-94040-CMC, version 02.00.13. Herndon, Virginia: Software Productivity Consortium.

1993 — *Consortium Requirements Engineering Guidebook,* SPC-92060-CMC, version 01.00.09. Herndon, Virginia: Software Productivity Consortium.

This page intentionally left blank.